Chapter 23:
Algorithms

Learning objectives

By the end of this chapter you should be able to:

- show understanding of linear and binary searching methods
- write a linear search algorithm
- write a binary search algorithm
- show understanding of the conditions necessary for the use of a binary search
- show understanding of how the performance of a binary search varies according to the number of data items
- show understanding of insertion sort and bubble sort methods
- show understanding that performance of a sorting routine may depend on the initial order of the data and the number of data items
- show understanding of and use Abstract Data Types (ADT)
- show how it is possible for ADTs to be implemented from another ADT
- describe the following ADTs and demonstrate how they can be implemented from appropriate built-in types or other ADTs: stack, queue, linked list, dictionary, binary tree
- write algorithms to:
  - implement an insertion sort
  - implement a bubble sort
  - find an item in each of the following: linked list, binary tree
  - insert an item into each of the following: stack, queue, linked list, binary tree
  - delete an item from each of the following: stack, queue, linked list
- show understanding that a graph is an example of an ADT
- describe the key features of a graph and justify its use for a given situation
- show understanding that different algorithms which perform the same task can be compared by using criteria such as time taken to complete the task and memory used
- show understanding of Big O notation for time and space complexity.

23.01 Linear search

In Chapter 13, we developed the algorithm for a linear search (Worked Example 13.02).

Discussion Point:

What were the essential features of a linear search?

TASK 23.01

Write program code for the linear search algorithm. Assume that the items to be searched are stored in a 1D array with n elements.

23.02 Bubble sort

In Chapter 13, we developed the algorithm for a bubble sort (Worked Example 13.03).

Discussion Point:

What were the essential features of a bubble sort?

TASK 23.02

Write program code for the most efficient bubble sort algorithm. Assume that the items to be sorted are stored in a 1D array with n elements.

23.03 Insertion sort

Imagine you have a number of cards with a different value printed on each card. How would you sort these cards into order of increasing value?

You can consider the pile of cards as consisting of a sorted part and an unsorted part. Place the unsorted cards in a pile on the table. Hold the sorted cards as a pack in your hand. To start with only the first (top) card is sorted. The card on the top of the pile on the table is the next card to be inserted. The last (bottom) card in your hand is your current card.

Figure 23.01 shows the sorted cards in your hand as blue and the pile of unsorted cards as white. The next card to be inserted is shown in red. Each column shows the state of the pile as the cards are sorted.

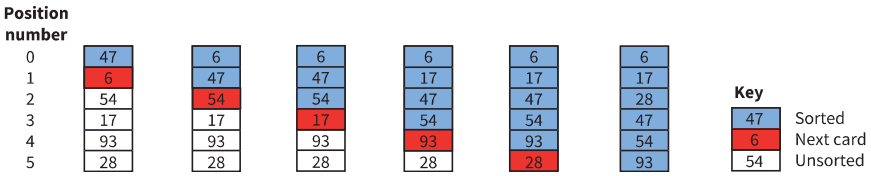


Figure 23.01 Sorting cards

Repeat the following steps until all cards in the unsorted pile have been inserted into the correct position.

1 Repeat until the card to be inserted has been placed in its correct position.

1.1 Compare the current card with the card to be inserted.

1.2 If the card to be inserted is greater than the current card, insert it below the current card.

1.3 Otherwise, if there is a card above the current card in your hand, make this your new current card.

1.4 If there is no new current card, place the card to be inserted at the top of the sorted pile.

What happens when you work through the sorted cards to find the correct position for the card to be inserted? In effect, as you consider the cards in your hand, you move the current card down a position. If the value of the card to be inserted is smaller than the last card you considered, then the card is inserted at the top of the pile (position 1).

This method is known as an insertion sort. It is a standard sort method.

We can write this algorithm using pseudocode. Assume the values to be sorted are stored in a 1D array, `List`:

```
FOR Pointer ← 1 TO NumberOfitems – 1
    ItemToBeInserted ← List[Pointer]
    CurrentItem ← Pointer – 1 // pointer to last item in sorted part of list
    WHILE (List[CurrentItem] > ItemToBeInserted) AND (CurrentItem > –1) DO
        List[CurrentItem + 1] ← List[CurrentItem] // move current item down
        CurrentItem ← CurrentItem – 1 // look at the item above
    ENDWHILE
    List[CurrentItem + 1] ← ItemToBeInserted // insert item
NEXT Pointer
```

TASK 23.03

1 Dry-run the insertion sort algorithm using a trace table. Assume the list consists of the following six items in the order given: 53, 21, 60, 18, 42, 19.

2 Write program code for the insertion sort algorithm. Assume that the items to be sorted are

stored in a 1D array with n elements.

Extension Question 23.01

Investigate the performances of the insertion sort and the bubble sort by:

- varying the initial order of the items
- increasing the number of items to be sorted.

23.04 Binary search

In Section 23.01 we considered the algorithm for a linear search. This is the only way we can systematically search an unordered list. However, if the list is ordered, then we can use a different technique.

Consider the following real-world example.

If you want to look up a word in a dictionary, you are unlikely to start searching for the word from the beginning of the dictionary. Suppose you are looking for the word ‘quicksort’. You look at the middle entry of the dictionary (approximately) and find the word ‘magnetic’. ‘quicksort’ comes after ‘magnetic’, so you look in the second half of the dictionary. Again, you look at the entry in the middle of this second half of the dictionary (approximately) and find the word ‘report’. ‘quicksort’ comes before ‘report’, so you look in the third quarter. You can keep looking at the middle entry of the part which must contain your word, until you find the word. If the word does not exist in the dictionary, you will have no entries in the dictionary left to find the middle of.

This method is known as a **binary search**. It is a standard method.

We can write this algorithm using pseudocode. Assume the values are sorted in ascending order and stored in a 1D array, `List` of size `MaxItems`.

```
Found ← FALSE
SearchFailed ← FALSE
First ← 0
Last ← MaxItems – 1 // set boundaries of search area
WHILE NOT Found AND NOT SearchFailed DO
    Middle ← (First + Last) DIV 2 // find middle of current search area
    IF List[Middle] = SearchItem
      THEN
        Found ← TRUE
      ELSE
        IF First >= Last // no search area left
          THEN
            SearchFailed ← TRUE
          ELSE
            IF List[Middle] > SearchItem
              THEN  // must be in first half
                Last ← Middle - 1  // move upper boundary
              ELSE  // must be in second half
                First ← Middle + 1  // move lower boundary
            ENDIF
        ENDIF
    ENDIF
ENDWHILE
IF Found = TRUE
  THEN
    OUTPUT Middle // output position where item was found
  ELSE
    OUTPUT "Item not present in array"
ENDIF
```

TASK 23.04

1 Dry-run the binary search algorithm using a trace table. Assume the list consists of the following 20 items in the order given: 7, 12, 19, 23, 27, 33, 37, 41, 45, 56, 59, 60, 62, 71, 75,

80, 84, 88, 92, 99.

2. Search for the value 60. How many times did you have to execute the `While` loop?
3. Dry-run the algorithm again, this time searching for the value 34. How many times did you have to execute the While loop?

Discussion Point:

Compare the binary-search algorithm with the linear-search algorithm. If the array contains n items, how many times on average do you need to test a value when using a binary search and how many times on average do you need to test a value when using a linear search? Can you describe how the search time varies with increasing n?

23.05 Abstract Data Types (ADTs)

In Chapter 13 (Section 13.07 to 13.10), we introduced ADTs using conceptual diagrams and how ADTs can be implemented using arrays.

23.06 Linked lists

Look back at Chapter 13 Section 13.10. Figure 13.25 shows an empty linked list. The `StartPointer` variable contains the Null pointer. The free list links all empty nodes.

We now code the basic operations discussed using the conceptual diagrams in Figures 13.11 to 13.16.

Create a new linked list

```
// NullPointer should be set to -1 if using array element with index 0
CONSTANT NullPointer = –1
// Declare record type to store data and pointer
TYPE ListNode
    DECLARE Data    : STRING
    DECLARE Pointer : INTEGER
ENDTYPE
DECLARE StartPointer : INTEGER
DECLARE FreeListPtr  : INTEGER
DECLARE List : ARRAY[0 : 6] OF ListNode

PROCEDURE InitialiseList
    StartPointer ← NullPointer       // set start pointer
    FreeListPtr ← 0                 // set starting position of free list
    FOR Index ← 0 TO 5              // link all nodes to make free list
        List[Index].Pointer ← Index + 1
    NEXT Index
    List [6].Pointer ← NullPointer   // last node of free list
ENDPROCEDURE
```

Insert a new node into an ordered linked list

```
PROCEDURE InsertNode(NewItem)
    IF FreeListPtr <> NullPointer
      THEN  // there is space in the array
        // take node from free list and store data item
        NewNodePtr ← FreeListPtr
        List[NewNodePtr].Data ← NewItem
        FreeListPtr ← List[FreeListPtr].Pointer
        // find insertion point
        ThisNodePtr ← StartPointer  // start at beginning of list
        PreviousNodePtr ← NullPointer
        WHILE ThisNodePtr <> NullPointer  // while not end of list
          AND List[ThisNodePtr].Data < NewItem DO
            PreviousNodePtr ← ThisNodePtr  // remember this node
            // follow the pointer to the next node
            ThisNodePtr ← List[ThisNodePtr].Pointer
        ENDWHILE
        IF PreviousNodePtr = StartPointer
          THEN  // insert new node at start of list
            List[NewNodePtr].Pointer ← StartPointer
            StartPointer ← NewNodePtr
          ELSE  // insert new node between previous node and this node
            List[NewNodePtr].Pointer ← List[PreviousNodePtr].Pointer
            List[PreviousNodePtr].Pointer ← NewNodePtr
```

```
        ENDIF
    ENDIF
ENDPROCEDURE
```

After three data items have been added to the linked list, the array contents are as shown in Figure 23.02.

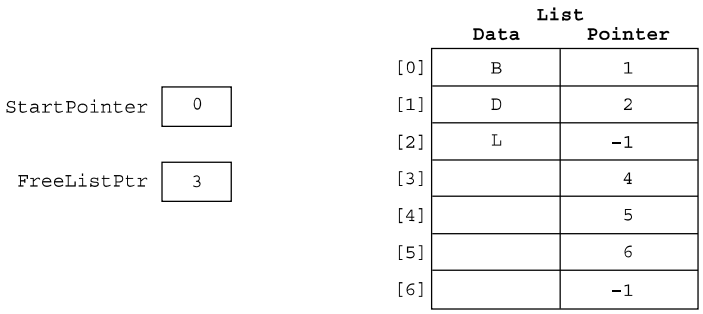


Figure 23.02 Linked list of three nodes and free list of four nodes

Find an element in an ordered linked list

```
FUNCTION FindNode(DataItem) RETURNS INTEGER  // returns pointer to node
    CurrentNodePtr ← StartPointer  // start at beginning of list
    WHILE CurrentNodePtr <> NullPointer  // not end of list
      AND List[CurrentNodePtr].Data <> DataItem DO // item not found
        // follow the pointer to the next node
        CurrentNodePtr ← List[CurrentNodePtr].Pointer
    ENDWHILE
    RETURN CurrentNodePtr  // returns NullPointer if item not found
ENDFUNCTION
```

Delete a node from an ordered linked list

```
PROCEDURE DeleteNode(DataItem)
    ThisNodePtr ← StartPointer             // start at beginning of list
    WHILE ThisNodePtr <> NullPointer       // while not end of list
      AND List[ThisNodePtr].Data <> DataItem DO // and item not found
        PreviousNodePtr ← ThisNodePtr  // remember this node
        // follow the pointer to the next node
        ThisNodePtr ← List[ThisNodePtr].Pointer
    ENDWHILE
    IF ThisNodePtr <> NullPointer  // node exists in list
      THEN
        IF ThisNodePtr = StartPointer // first node to be deleted
          THEN
            // move start pointer to the next node in list
            StartPointer ← List[StartPointer].Pointer
          ELSE
            // it is not the start node;
            // so make the previous node's pointer point to
            // the current node's 'next' pointer; thereby removing all
            // references to the current pointer from the list
            List[PreviousNodePtr].Pointer ← List[ThisNodePtr].Pointer
        ENDIF
        List[ThisNodePtr].Pointer ← FreeListPtr
        FreeListPtr ← ThisNodePtr
    ENDIF
```

```
ENDPROCEDURE
```

Access all nodes stored in the linked list

```
PROCEDURE OutputAllNodes
    CurrentNodePtr ← StartPointer // start at beginning of list
    WHILE CurrentNodePtr <> NullPointer DO // while not end of list
        OUTPUT List[CurrentNodePtr].Data
        // follow the pointer to the next node
        CurrentNodePtr ← List[CurrentNodePtr].Pointer
    ENDWHILE
ENDPROCEDURE
```

TASK 23.05

Convert the pseudocode for the linked-list handling subroutines to program code. Incorporate the subroutines into a program and test them.

Note that a stack ADT and a queue ADT can be treated as special cases of linked lists. The linked list stack only needs to add and remove nodes from the front of the linked list (see Section 23.08). The linked list queue only needs to add nodes to the end of the linked list and remove nodes from the front of the linked list (see Section 23.09).

23.07 Binary trees

In the real world, we draw tree structures to represent hierarchies. For example, we can draw a family tree showing ancestors and their children. A binary tree is different to a family tree because each node can have at most two ‘children’.

In computer science binary trees are used for different purposes. In Chapter 20 (Section 20.06), you saw the use of a binary tree as a syntax tree. In this chapter, you will use an ordered binary tree ADT (such as the one shown in Figure 23.03) as a binary search tree.

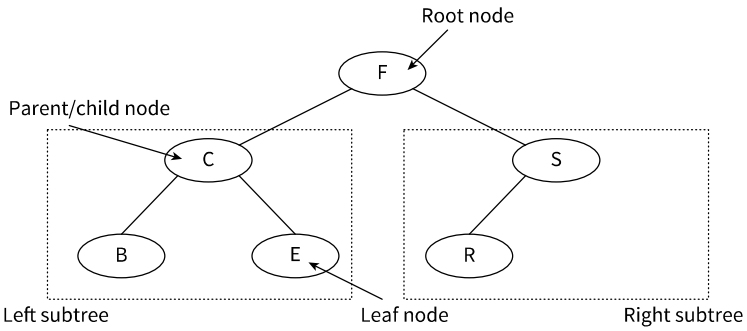


Figure 23.03 Conceptual diagram of an ordered binary tree

Nodes are added to an ordered binary tree in a specific way:

> Start at the root node as the current node.
>
> Repeat
>
> > If the data value is greater than the current node’s data value, follow the right branch.
> >
> > If the data value is smaller than the current node’s data value, follow the left branch.
>
> Until the current node has no branch to follow.
>
> Add the new node in this position.

For example, if we want to add a new node with data value D to the binary tree in Figure 23.03, we execute the following steps.

1. Start at the root node.
2. D is smaller than F, so turn left.
3. D is greater than C, so turn right.
4. D is smaller than E, so turn left.
5. There is no branch going left from E, so we add D as a left child from E (see Figure 23.04).

This type of tree has a special use as a search tree. Just like the binary search applied to an ordered linear list, the binary search tree gives the benefit of a faster search than a linear search or searching a linked list. The ordered binary tree also has a benefit when adding a new node: other nodes do not need to be moved, only a left or right pointer needs to be added to link the new node into the existing tree.

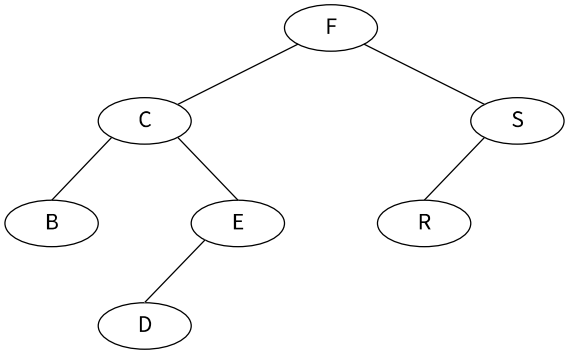


Figure 23.04 Conceptual diagram of adding a node to an ordered binary tree

We can store the binary tree in an array of records (see Figure 23.05). One record represents a node and consists of the data and a left pointer and a right pointer. Unused nodes are linked together to form a free list.

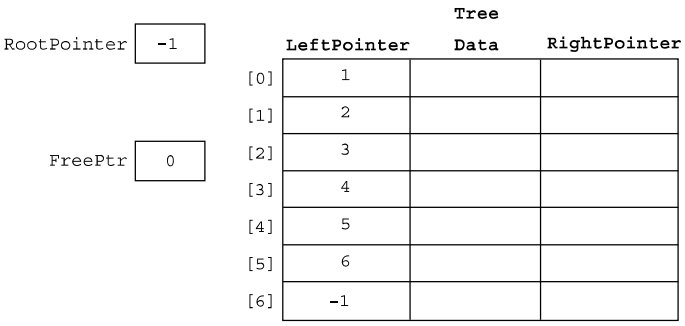


Figure 23.05 Binary tree before any nodes are inserted

Create a new binary tree

```
// NullPointer should be set to -1 if using array element with index 0
CONSTANT NullPointer = –1
// Declare record type to store data and pointers
TYPE TreeNode
    DECLARE Data : STRING
    DECLARE LeftPointer : INTEGER
    DECLARE RightPointer : INTEGER
ENDTYPE
DECLARE RootPointer : INTEGER
DECLARE FreePtr : INTEGER
DECLARE Tree : ARRAY[0 : 6] OF TreeNode
PROCEDURE InitialiseTree
    RootPointer ← NullPointer  // set start pointer
    FreePtr ← 0                // set starting position of free list
    FOR Index ← 0 TO 5         // link all nodes to make free list
        Tree[Index].LeftPointer ← Index + 1
    NEXT Index
    Tree [6].LeftPointer ← NullPointer // last node of free list
ENDPROCEDURE
```

Insert a new node into a binary tree

```
PROCEDURE InsertNode(NewItem)
    IF FreePtr <> NullPointer
      THEN  // there is space in the array
        // take node from free list, store data item, set null pointers
        NewNodePtr ← FreePtr
        FreePtr ← Tree[FreePtr].LeftPointer
        Tree[NewNodePtr].Data ← NewItem
        Tree[NewNodePtr].LeftPointer ← NullPointer
        Tree[NewNodePtr].RightPointer ← NullPointer
        // check if empty tree
        IF RootPointer = NullPointer
          THEN  // insert new node at root
            RootPointer ← NewNodePtr
          ELSE  // find insertion point
```

```
            ThisNodePtr ← RootPointer   // start at the root of the tree
            WHILE ThisNodePtr <> NullPointer DO  // while not a leaf node
                PreviousNodePtr ← ThisNodePtr  // remember this node
                IF Tree[ThisNodePtr].Data > NewItem
                  THEN  // follow left pointer
                    TurnedLeft ← TRUE
                    ThisNodePtr ← Tree[ThisNodePtr].LeftPointer
                  ELSE // follow right pointer
                    TurnedLeft ← FALSE
                    ThisNodePtr ← Tree[ThisNodePtr].RightPointer
                ENDIF
            ENDWHILE
            IF TurnedLeft = TRUE
              THEN
                Tree[PreviousNodePtr].LeftPointer ← NewNodePtr
              ELSE
                Tree[PreviousNodePtr].RightPointer ← NewNodePtr
            ENDIF
        ENDIF
    ENDIF
ENDPROCEDURE
```

Find a node in a binary tree

```
FUNCTION FindNode(SearchItem) RETURNS INTEGER  // returns pointer to node
    ThisNodePtr ← RootPointer     // start at the root of the tree
    WHILE ThisNodePtr <> NullPointer   // while a pointer to follow
      AND Tree[ThisNodePtr].Data <> SearchItem DO // and search item not found
        IF Tree[ThisNodePtr].Data > SearchItem
          THEN  // follow left pointer
            ThisNodePtr ← Tree[ThisNodePtr].LeftPointer
          ELSE // follow right pointer
            ThisNodePtr ← Tree[ThisNodePtr].RightPointer
        ENDIF
    ENDWHILE
    RETURN ThisNodePtr // will return null pointer if search item not found
ENDFUNCTION
```

TASK 23.06

Write program code to implement a binary search tree.

23.08 Stacks

In Chapter 13 (Section 13.08) we looked at the conceptual data structure of a stack. A stack can be implemented using a 1D array. Figure 23.06 shows a stack containing four data items. For conceptual reasons the array elements are numbered from the bottom up.

Note that `BaseOfStackPointer` will always point to element 0 of the array. `TopOfStackPointer` will vary. It will increase when an item is pushed onto the stack and it will decrease when an item is popped off the stack. When the stack is empty, `TopOfStackPointer` will have the value–1.

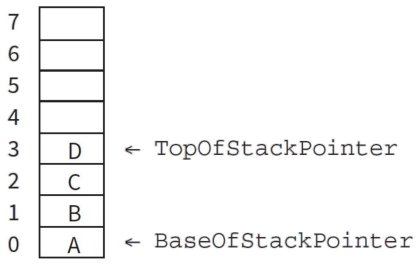


Figure 23.06 A stack

Create a new stack

```
// NullPointer should be set to -1 if using array element with index 0
CONSTANT EMPTYSTRING = ""
CONSTANT NullPointer = –1
CONSTANT MaxStackSize = 8
DECLARE BaseOfStackPointer : INTEGER
DECLARE TopOfStackPointer : INTEGER
DECLARE Stack : ARRAY[1 : MaxStackSize – 1] OF STRING

PROCEDURE InitialiseStack
    BaseOfStackPointer ← 0             // set base of stack pointer
    TopOfStackPointer ← NullPointer    // set top of stack pointer
ENDPROCEDURE
```

Push an item onto the stack

```
PROCEDURE Push(NewItem)
    IF TopOfStackPointer < MaxStackSize – 1
      THEN     // there is space on the stack
               // increment top of stack pointer
        TopOfStackPointer ← TopOfStackPointer + 1
               // add item to top of stack
        Stack[TopOfStackPointer] ← NewItem
    ENDIF
ENDPROCEDURE
```

Pop an item off the stack

```
FUNCTION Pop()
    DECLARE Item : STRING
    Item ← EMPTYSTRING
    IF TopOfStackPointer > NullPointer
      THEN     // there is at least one item on the stack
               // pop item off the top of the stack
        Item ← Stack[TopOfStackPointer]
               // decrement top of stack pointer
```

```
        TopOfStackPointer ← TopOfStackPointer – 1
    ENDIF
    RETURN Item
ENDFUNCTION
```

TASK 23.07

1. Write a program to implement the above pseudocode subroutines. Add a menu to test the subroutines.
2. Write a program to implement a stack as a linked list. Note that the adding and removing of nodes is much simpler than for an ordered linked list.

23.09 Queues

In Chapter 13 (Section 13.09) we looked at the conceptual data structure of a queue. A queue can be implemented using a 1D array. Figure 23.07 shows a queue containing five data items.

`FrontOfQueuePointer` always points to the first element in the queue, that is the next element to be taken from the queue. `EndOfQueuePointer` always points to the last element in the queue. Before another element joins the queue, the `EndOfQueuePointer` is incremented. Note that when adjusting either pointer the possibility of wrap-round has to be tested.

To make it easier to test whether the queue is empty or full, a counter variable can be used.

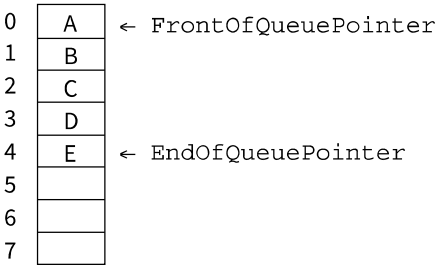


Figure 23.07 A queue before wrap-round

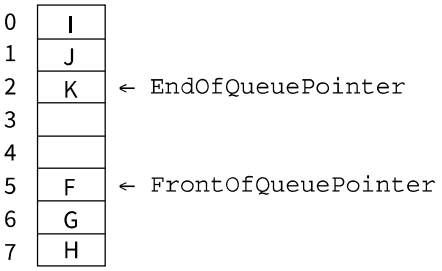


Figure 23.08 A queue with wrap-round

Create a new queue

```
// NullPointer should be set to -1 if using array element with index 0
CONSTANT EMPTYSTRING = ""
CONSTANT NullPointer = -1
CONSTANT MaxQueueSize = 8
DECLARE FrontOfQueuePointer : INTEGER
DECLARE EndOfQueuePointer : INTEGER
DECLARE NumberInQueue : INTEGER
DECLARE Queue : ARRAY[0 : MaxQueueSize – 1] OF STRING
PROCEDURE InitialiseQueue
    FrontOfQueuePointer ← NullPointer   // set front of queue pointer
    EndOfQueuePointer ← NullPointer     // set end of queue pointer
    NumberInQueue ← 0                   // no elements in queue
ENDPROCEDURE
```

Add an item to the queue

```
PROCEDURE AddToQueue(NewItem)
    IF NumberInQueue < MaxQueueSize
      THEN     // there is space in the queue
               // increment end of queue pointer
        EndOfQueuePointer ← EndOfQueuePointer + 1
               // check for wrap-round
        IF EndOfQueuePointer > MaxQueueSize – 1
```

```
          THEN     // wrap to beginning of array
            EndOfQueuePointer ← 0
                    // add item to end of queue
        ENDIF
        Queue[EndOfQueuePointer] ← NewItem
         // increment counter
         NumberInQueue ← NumberInQueue + 1
    ENDIF
ENDPROCEDURE
```

Remove an item from the queue

```
FUNCTION RemoveFromQueue()
    DECLARE Item : STRING
    Item ← EMPTYSTRING
    IF NumberInQueue > 0
      THEN // there is at least one item in the queue
           // remove item from the front of the queue
        Item ← Queue[FrontOfQueuePointer]
        NumberInQueue ← NumberInQueue – 1
        IF NumberInQueue = 0
          THEN // if queue empty, reset pointers
            CALL InitialiseQueue
          ELSE
            // increment front of queue pointer
            FrontOfQueuePointer ← FrontOfQueuePointer + 1
            // check for wrap-round
            IF FrontOfQueuePointer > MaxQueueSize – 1
              THEN // wrap to beginning of array
              FrontOfQueuePointer ← 0
            ENDIF
        ENDIF
    ENDIF
    RETURN Item
ENDFUNCTION
```

TASK 23.08

1. Write a program to implement the above pseudocode subroutines. Add a menu to test the subroutines.
2. Write program code to implement a queue as a linked list. You may find it helpful to introduce another pointer that always points to the end of the queue. You will need to update it when you add a new node to the queue.

23.10 Graphs

In Computer Science a graph is an ADT consisting of vertices (nodes) and edges. Graphs are used to record relationships between things. For uses of graphs in AI see Chapter 22 Section 22.02. A simple graph is shown in Figure 23.09. It represents a small part of the London Underground map shown in Figure 12.02 in Chapter 12.

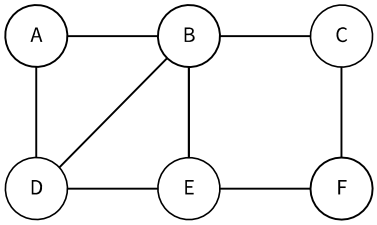


Figure 23.09 Graph showing part of the London Underground map

The vertices labelled A to F are the underground stations and the edges represent train lines connecting the stations. For example, you can take a train directly from B to D. To get from B to F, you have to travel via C or E. Two vertices connected by an edge are known as neighbours.

A labelled (weighted) graph has edges with values representing something. In our example, we can add weights to Figure 23.10 to show the time it takes to travel between stations:

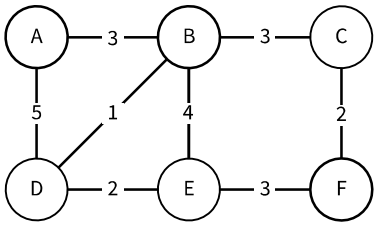


Figure 23.10 Weighted graph showing travelling times between stations

Graphs can be directed or undirected (as in Figure 23.10). Travel times may vary depending on the direction of travel. This can be shown in a directed graph (see Figure 23.11).

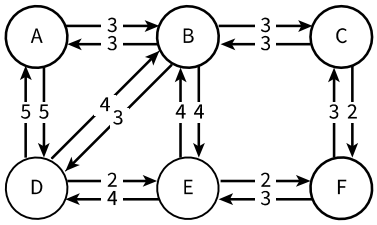


Figure 23.11 Directed graph

Sometimes one direction may not be available. For example, if the line from B to D is blocked, this could be represented as shown in Figure 23.12.

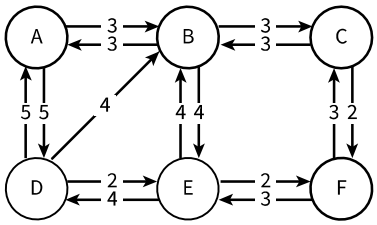


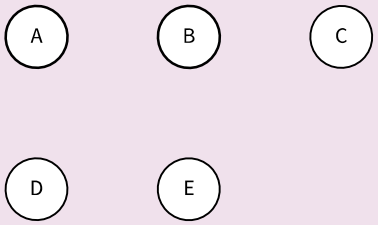
Figure 23.12 Directed graph

We can use a graph to plan a journey. Using Figure 23.12, we want to travel from C to D, so we can either use the route C B A D (11 minutes) or the route C F E D (9 minutes). For the return journey we can use D B C (7 minutes).

TASK 23.09

Draw the labelled edges for a directed graph of a road system where:

- A to B is 2 km, and it is a one-way system
- D to A is 1 km, and is one-way
- B to D is 3 km and is one-way
- B to C is 4 km
- C to E is 7 km
- D to E is 5 km.



To implement a graph, we can use an adjacency matrix or an adjacency list.

An adjacency matrix stores the relationship between every vertex to all other vertices. For an unweighted graph, a 1 represents an edge, a 0 no edge. When weights are to be recorded, the weight replaces the 1. Instead of a 0, we use the infinity symbol ∞.

	A	B	C	D	E	F
A	0	1	0	1	0	0
B	1	0	1	1	1	0
C	0	1	0	0	0	1
D	1	1	0	0	1	0
E	0	1	0	1	0	1
F	0	0	1	0	1	0

Table 23.01

Adjacency matrix for Figure 23.09

	A	B	C	D	E	F
A	∞	3	∞	5	∞	∞
B	3	∞	3	∞	4	∞
C	∞	3	∞	∞	∞	2
D	5	4	∞	∞	2	∞
E	∞	4	∞	4	∞	2
F	∞	∞	3	∞	3	∞

Table 23.02

Adjacency matrix for Figure 23.12

An adjacency list stores the relationship between every vertex to all relevant vertices. An entry is made only when there is an edge between two vertices. For a weighted graph, the connection as well as the

weight is stored in the list.

	connected to
A	B, D
B	A, C, D, E
C	B, F
D	A, B, E
E	B, D, F
F	C, E

Table 23.03

Adjacency list for Figure 23.09

	connected to
A	B, 3; D, 5
B	A, 3; C, 3; E, 4
C	B, 3; F, 2
D	A, 5; B, 4; E, 2
E	B, 4; D, 4; F, 2
F	C, 3; E, 3

Table 23.04

Adjacency list for Figure 23.12

TASK 23.10

Construct an adjacency matrix and an adjacency list to represent your graph from Task 23.09.

23.11 Hash tables

If we want to store records in an array and have direct access to records, we can use the concept of a hash table.

The idea behind a hash table is that we calculate an address (the array index) from the key value of the record and store the record at this address. When we search for a record, we calculate the address from the key and go to the calculated address to find the record. Calculating an address from a key is called ‘hashing’.

Finding a hashing function that will give a unique address from a unique key value is very difficult. If two different key values hash to the same address this is called a ‘collision’. There are different ways to handle collisions:

- chaining: create a linked list for collisions with start pointer at the hashed address
- using overflow areas: all collisions are stored in a separate overflow area, known as ‘closed hashing’
- using neighbouring slots: perform a linear search from the hashed address to find an empty slot, known as ‘open hashing’.

WORKED EXAMPLE 23.01

Calculating addresses in a hash table

Assume we want to store customer records in a 1D array `HashTable[0 : n]`. Each customer has a unique customer ID, an integer in the range 10001 to 99999.

We need to design a suitable hashing function. The result of the hashing function should be such that every index of the array can be addressed directly. The simplest hashing function gives us addresses between 0 and n:

```
FUNCTION Hash(Key) RETURNS INTEGER
    Address ← Key MOD(n + 1)
    RETURN Address
ENDFUNCTION
```

For illustrative purposes, we choose n to be 9. Our hashing function is:

```
Index ← CustomerID MOD 10
```

We want to store records with customer IDs: 45876, 32390, 95312, 64636, 23467. We can store the first three records in their correct slots, as shown in Figure 23.13.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
32390		95312				45876			

Figure 23.13 A hash table without collisions

The fourth record key (64636) also hashes to index 6. This slot is already taken; we have a collision. If we store our record here, we lose the previous record. To resolve the collision, we can choose to store our record in the next available space, as shown in Figure 23.14.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
32390		95312				45876	64636		

Figure 23.14 A hash table with a collision resolved by open hashing

The fifth record key (23467) hashes to index 7. This slot has been taken up by the previous record, so again we need to use the next available space (Figure 23.15).

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
32390		95312				45876	64636	23467	

Figure 23.15 A hash table with two collisions resolved by open hashing

When searching for a record, we need to allow for these out-of-place records. We know if the record we are searching for does not exist in the hash table when we come across an unoccupied slot.

We will now develop algorithms to insert a record into a hash table and to search for a record in the hash table using its record key.

The hash table is a 1D array `HashTable[0 : Max] OF Record`.

The records stored in the hash table have a unique key stored in field `Key`.

Insert a record into a hash table

```
PROCEDURE Insert(NewRecord)
    Index ← Hash(NewRecord.Key)
    WHILE HashTable[Index] NOT empty DO
        Index ← Index + 1 // go to next slot to check if empty
        IF Index > Max // beyond table boundary?
          THEN // wrap around to beginning of table
            Index ← 0
        ENDIF
    ENDWHILE
    HashTable[Index] ← NewRecord
ENDPROCEDURE
```

Find a record in a hash table

```
FUNCTION FindRecord(SearchKey) RETURNS Record
    Index ← Hash(SearchKey)
    WHILE (HashTable[Index].Key <> SearchKey)
      AND (HashTable[Index] NOT empty) DO
        Index ← Index + 1 // go to next slot
        IF Index > Max // beyond table boundary?
          THEN // wrap around to beginning of table
            Index ← 0
        ENDIF
    ENDWHILE
    IF HashTable[Index] NOT empty // if record found
      THEN
        RETURN HashTable[Index] // return the record
    ENDIF
ENDFUNCTION
```

23.12 Dictionaries

A real-world dictionary is a collection of key–value pairs. The key is the term you use to look up the required value. For example, if you use an English–French dictionary to look up the English word 'book', you will find the French equivalent word 'livre'. A real-world dictionary is organised in alphabetical order of keys.

An ADT dictionary in computer science is implemented using a hash table (see Section 23.11), so that a value can be looked up using a direct-access method.

Python, VB.NET and Java have a built-in ADT dictionary class.

Here are some examples of Python dictionaries:

```
EnglishFrench = {} # empty dictionary
EnglishFrench["book"] = "livre" # add a key-value pair to the dictionary
EnglishFrench["pen"] = "stylo"
print(EnglishFrench["book"]) # access a value in the dictionary
# alternative method of setting up a dictionary
ComputingTerms = {"Boolean" : "can be TRUE or FALSE", "Bit" : "0 or 1"}
print(ComputingTerms["Bit"])
```

Here are some examples of VB dictionaries:

```
Dim EnglishFrench As New Dictionary(Of String, String)
EnglishFrench.Add("book", "livre")
EnglishFrench.Add("pen", "stylo")
Console.WriteLine(EnglishFrench.Item("book"))
Dim ComputingTerms As New Dictionary(Of String, String)
ComputingTerms.Add("Boolean", "can be TRUE or FALSE")
ComputingTerms.Add("Bit", "0 or 1")
Console.WriteLine(ComputingTerms.Item("Bit"))
Console.ReadLine()
```

Here are some examples of Java dictionaries (the Dictionary class is obsolete, use HashMap instead):

```
import java.util.Map;
import java.util.HashMap;
    Map<String, String> englishFrench = new HashMap<String, String>();
    englishFrench.put("book", "livre");
    englishFrench.put("pen", "stylo");
    System.out.println(englishFrench.get("book"));
    Map<String, String>  computingTerms = new HashMap<String, String>();
    computingTerms.put("Boolean", "can be TRUE or FALSE");
    computingTerms.put("Bit", "0 or 1");
    System.out.println(computingTerms.get("Bit"));
```

There are many built-in functions for Python, VB and Java dictionaries. These are beyond the scope of this book. However, we need to understand how dictionaries are implemented. The following pseudocode shows how to create a new dictionary.

```
TYPE DictionaryEntry
    DECLARE Key   : STRING
    DECLARE Value : STRING
ENDTYPE
DECLARE EnglishFrench : ARRAY[0 : 999] OF DictionaryEntry  // empty dictionary
```

TASK 23.11

Write pseudocode to:

- insert a key–value pair into a dictionary
- look up a value in a dictionary.

Use the hashing function from Worked Example 23.01.

23.13 Big O notation

A problem can be solved in different ways, with different algorithms. Clearly, we want to use time and memory efficiently. A way of comparing the efficiency of algorithms has been devised using order of growth as a function of the size of the input. Big O notation is used to classify algorithms according to how their running time (or space requirements) grows as the input size grows. The letter O is used because the growth rate of a function is also referred to as 'order of the function'. The worst-case scenario is used when calculating the order of growth for very large data sets.

Consider the linear search algorithm in Chapter 13, Worked Example 13.02. The worst case scenario is that the item searched for is the last item in the list. The longer the list, the more comparisons have to be made. If the list is twice as long, twice as many comparisons have to be made. Generally, we can say the order of growth is linear. We write this as O(n), where n is the size of the data set.

Consider the bubble sort algorithm for the worst case scenario.

```
Unsorted ← n – 1
FOR i ← 0 TO n – 2
    FOR j ← 0 TO Unsorted - 1
        IF MyList[j] > MyList[j + 1]
          THEN
            Temp ← MyList[j]
            MyList[j] ← MyList[j + 1]
            MyList[j + 1] ← Temp
        ENDIF
    NEXT j
    Unsorted ← Unsorted - 1
NEXT i
```

The basic operation for this algorithm is the comparison `IF MyList[j] > MyList[j + 1]`

n	Number of comparisons	
1	0	
2	1	
3	3	= 1 + 2
4	6	= 1 + 2 + 3
5	10	= 1 + 2 + 3 + 4
6	15	= 1 + 2 + 3 + 4 + 5

Table 23.05 Number of comparisons in bubble sort

We can see that the total number of comparisons is the sum of the first (n – 1) whole numbers. This leads us to the formula:

½ * n * (n – 1) = ½ * (n^2 – n)

For very large n we can disregard all factors except the largest and we get n^2. So the order of growth is n^2.

Consider the binary search algorithm in Section 23.04. With each iteration this algorithm halves the number of values in the data set. This iterative halving of data sets produces a growth curve that peaks at the beginning and slowly flattens out as the size of the data sets increase. This type of algorithm is described as O($\log_2 n$)

Table 23.06 shows a summary of standard algorithms and their order of growth (time complexity).

Order of growth	Example	Explanation

$O(1)$	`FUNCTION GetFirstItem(List : ARRAY) RETURN List[1]`	The complexity of the algorithm does not change regardless of data set size
$O(n)$	Linear search Bubble sort performed on an already sorted list	Linear growth
$O(\log_2 n)$	Binary search	The total time taken increases as the data set size increases, but each comparison halves the data set. So the time taken increases by smaller amounts and approaches constant time.
$O(n^2)$	Bubble sort Insertion sort	Polynomial growth Common with algorithms that involve nested iterations over the data set
$O(n^3)$		Polynomial growth Deeper nested iterations will result in $O(n^3)$, $O(n^4)$, ...
$O(2^n)$	Recursive calculation of Fibonacci numbers	Exponential growth

Table 23.06 Order of growth (time complexity) for data input set of size n

Space complexity refers to the amount of memory required for a growing number of values n in the dataset.

The ADT operations discussed in this chapter have space complexity O(n). This means that they only take the space required for the data set. Bubble sort and insertion sort have space complexity of O(1). This means they do not require extra memory for sorting larger lists.

Reflection Point:

List the standard algorithms you have met in this chapter. Can you give the essential features of each of these?

Summary

- Standard algorithms include bubble sort, insertion sort, linear search and binary search.
- Abstract data types (ADTs) include stacks, queues, linked lists, binary trees, hash tables, dictionaries and graphs.
- Basic operations required for an ADT include creating an ADT and inserting, finding or deleting an element of an ADT.
- Time taken and space used can be measured using Big O notation.

Exam-style Questions

1 **a** Complete the algorithm for a binary search function `FindName`.

The data being searched is stored in the array `Names[0 : 50]`.

The name to be searched for is passed as a parameter.

```
FUNCTION FindName(s : STRING) RETURNS INTEGER
    Index ← -1
    First ← 0
    Last ← 50
    WHILE (Last >= First) AND ..... DO
        Middle ← (First + Last) DIV 2
        IF Names[Middle] = s
          THEN
            Index ← Middle
          ELSE
            IF .....
             THEN
               Last ← Middle + 1
             ELSE
                .....
            ENDIF
        ENDIF
    ENDWHILE
ENDFUNCTION
```

[3]

b The binary search does not work if the data in the array being searched is ..... [1]

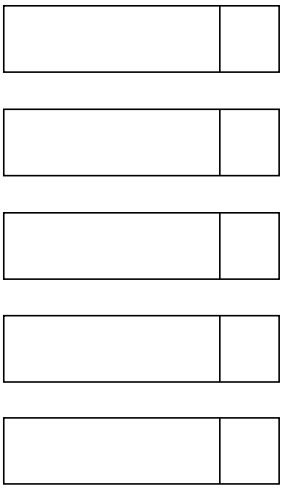
c What does the function `FindName` return when:

i the name searched for exists in the array?

ii the name searched for does not exist in the array? [2]

2 A queue Abstract Data Type (ADT) is to be implemented as a linked list of nodes. Each node is a record, consisting of a data field and a pointer field. The queue ADT also has a `FrontOfQueue` pointer and an `EndOfQueue` pointer associated with it. The possible queue operations are: `JoinQueue` and `LeaveQueue`.

a **i** Add labels to the diagram to show the state of the queue after three data items have been added to the queue in the given order: Apple, Pear, Banana.



[5]

ii Add labels to the diagram to show how the unused nodes are linked to form a list of free

nodes. This list has a `StartOfFreeList` pointer associated with it. [2]

b **i** Write program code to declare the record type `Node`. [3]

ii Write program code to create an array `Queue` with 50 records of type `Node`. Your solution should link all nodes and initialise the pointers `FrontOfQueue, EndOfQueue` and `StartOfFreeList`. [7]

c The pseudocode algorithm for the queue operation `JoinQueue` is written as a procedure with the header:

```
PROCEDURE JoinQueue(NewItem)
```

where `NewItem` is the new value to be added to the queue. The procedure uses the variables shown in the following identifier table:

Identifier	Data type	Description
`NullPointer`	`INTEGER`	Constant set to –1
		Array to store queue data
	`STRING`	Value to be added
		Pointer to next free node in array
		Pointer to first node in queue
		Pointer to last node in queue
		Pointer to node to be added

i Complete the identifier table. [7]

ii Complete the pseudocode using the identifiers from the table in **part (c) (i)**. [6]

```
PROCEDURE JoinQueue(NewItem : STRING)
    // Report error if no free nodes remaining
    IF StartOfFreeList = ............................
      THEN
        Report Error
      ELSE
        // new data item placed in node at start of free list
        NewNodePointer ← StartOfFreeList
        Queue[NewNodePointer].Data ← NewItem
        // adjust free list pointer
        StartOfFreeList ← Queue[NewNodePointer].Pointer
        Queue[NewNodePointer].Pointer ← NullPointer
        // if first item in queue then adjust front of queue pointer
        IF FrontOfQueue = NullPointer
          THEN
            ..............................
            ..............................
        ENDIF
        // new node is new end of queue
        Queue[.....................].Pointer ← .....................
        EndOfQueue ← ..........................................
    ENDIF
ENDPROCEDURE
```

3 A program is required that sorts a list of words into alphabetical order. The list of words is supplied as a text file.

a Write a program to declare a string array, `WordList`, that can hold 500 elements. Initialise the array so all elements contain the empty string. [3]

b Write a procedure, `OutputList`, to output all elements in index order. [3]

c Write a procedure, `LoadWords`, that asks the user for a filename and reads the contents of the text file, storing each line of text (word) in a separate array element. The procedure should output a relevant error message if:

- the file doesn't exist
- the array is full. [7]

d Write a procedure, `SortWords`, to perform a bubble sort on all non-empty array elements, so that the words are in alphabetical order. [7]

e Write program code to call `LoadWords`, then `OutputList`, followed by `SortWords` and then `OutputList` again. [3]

f Test your program by running it first with a non-existing file, and then with a text file containing 20 words in random order.

Make screenshots of your test runs that show your code works correctly. [2]

Chapter 24:
Recursion

Learning objectives

By the end of this chapter you should be able to:

- show understanding of the essential features of recursion
- show understanding of how recursion is expressed in a programming language
- trace recursive algorithms
- write recursive algorithms
- show understanding of when the use of recursion is beneficial
- show awareness of what a compiler has to do to translate recursive programming code, including the use of stacks and unwinding.

24.01 Concept of recursion

In mathematical logic and computer science, a function or procedure is said to be a **recursive routine** if it is defined in terms of itself.

The classic mathematical example is the factorial function, n!, which is defined in Figure 24.01. This definition holds for all positive whole numbers.

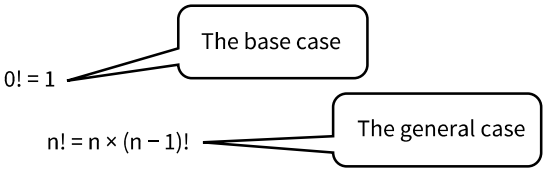


Figure 24.01 Mathematical definition of the factorial function

Figure 24.02 shows expressions of the factorial function for the first four numbers.

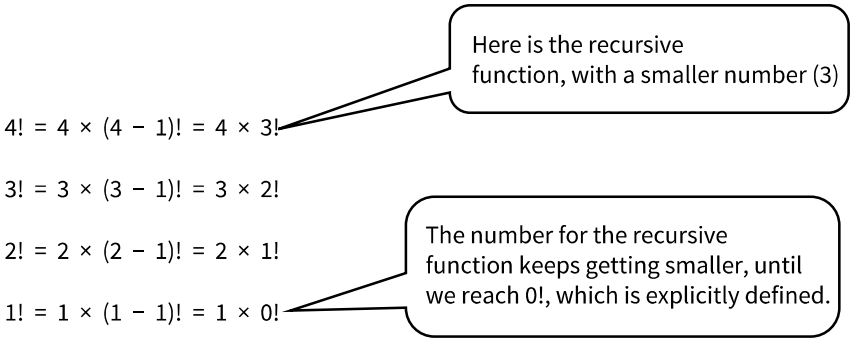

Here is the recursive function, with a smaller number (3)
4! = 4 × (4 − 1)! = 4 × 3!
3! = 3 × (3 − 1)! = 3 × 2!
2! = 2 × (2 − 1)! = 2 × 1!
1! = 1 × (1 − 1)! = 1 × 0!
The number for the recursive function keeps getting smaller, until we reach 0!, which is explicitly defined.

Figure 24.02 Expressions of the factorial function

Because 0! = 1:

$$4! = 4 \times 3 \times 2 \times 1 \times 1 = 24$$

Recursive solutions have a **base case** and a **general case**. The base case gives a result without involving the general case. The general case is defined in terms of the definition itself. It is very important that the general case must come closer to the base case with each recursion, for any starting point.

24.02 Programming a recursive subroutine

WORKED EXAMPLE 24.01

Coding the factorial function

We could program the function `Factorial` iteratively, using a loop:

```
FUNCTION Factorial(n : INTEGER) RETURNS INTEGER
    Result ← 1
    FOR i ← 1 TO n
        Result ← Result * i
    NEXT i
    RETURN Result
ENDFUNCTION
```

Alternatively, we can define the function `Factorial` recursively (Figure 24.03).

```
FUNCTION Factorial(n : INTEGER) RETURNS INTEGER
    IF n = 0
      THEN
        Result ← 1
      ELSE
        Result ← n * Factorial(n - 1)
    ENDIF
    RETURN Result
ENDFUNCTION
```

This is the base case

This is the general case

This is the recursive call

Figure 24.03 The factorial function defined recursively

The recursive pseudocode resembles the original mathematical definition of the factorial function. The dry run in Figure 24.05 (Section 24.03) shows how this works.

Discussion Point:

Carefully examine the two solutions to the factorial function. What happens if the iterative function is called with parameter 0? What happens if the recursive function is called with parameter 0? What changes would need to be made so the mathematical definition holds for all values of n?

When writing a recursive subroutine, there are three rules you must observe. A recursive subroutine must:

- have a base case
- have a general case
- reach the base case after a finite (limited) number of calls to itself.

TASK 24.01

Write program code to implement the recursive algorithm for the factorial function.

Question 24.01

What happens when the function is called with `Factorial(-2)`? Which rule is not satisfied?

WORKED EXAMPLE 24.02

Coding a recursive procedure

Consider a procedure to count down from a given integer. We can write the solution as an iterative algorithm:

```
PROCEDURE CountDownFrom(n : INTEGER)
   FOR i ← n DOWNTO 0
       OUTPUT i
   NEXT i
ENDPROCEDURE
```

We can also write the solution as a recursive algorithm. Consider what happens after the first value has been output. The remaining numbers follow the same pattern of counting down from the next smaller value. The base case is when n reaches 0. 0 will be output but no further numbers. The general case is outputting n and then counting down from (n – 1). This can be written using pseudocode:

```
PROCEDURE CountDownFrom(n : INTEGER)
    OUTPUT n
    IF n > 0
      THEN
        CALL CountDownFrom(n – 1)
    ENDIF
ENDPROCEDURE
```

24.03 Tracing a recursive subroutine

Tracing a recursive procedure

Dry-running the recursive procedure from Worked Example 24.02, we can complete a trace table as shown in Table 24.01.

Call number	Procedure call	OUTPUT	n > 0
1	`CountDownFrom(3)`	3	TRUE
2	`CountDownFrom(2)`	2	TRUE
3	`CountDownFrom(1)`	1	TRUE
4	`CountDownFrom(0)`	0	FALSE

Table 24.01 Trace table for `CALL CountDownFrom(3)`

It is more complex to trace a subroutine that contains statements to execute after the recursive call. Look at the slightly modified algorithm:

```
PROCEDURE CountUpTo(n : INTEGER)
    IF n > 0
      THEN
        CALL CountUpTo(n – 1)
    ENDIF
    OUTPUT n
ENDPROCEDURE
```

Note that the statements after `CALL CountUpTo(n – 1)` are not executed until control returns to this statement as the recursive calls unwind.

What is the effect of moving the `OUTPUT` statement to the end of the procedure? Figure 24.04 traces the execution of `CALL CountUpTo(3)`

Call number	Procedure call	n > 0	OUTPUT
1	`CountUpTo(3)`	TRUE	
2	`CountUpTo(2)`	TRUE	
3	`CountUpTo(1)`	TRUE	
4	`CountUpTo(0)`	FALSE	0
(3)	`CountUpTo(1)`	TRUE	1
(2)	`CountUpTo(2)`	TRUE	2
(1)	`CountUpTo(3)`	TRUE	3

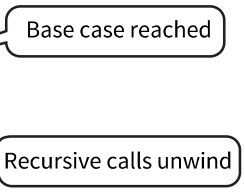


Figure 24.04 Trace table for `CALL CountUpTo(3)`

When the base case is reached, the fourth call of the procedure is complete and the procedure is exited. Control then passes back to the third call and so on. Note how we show the trace as the recursive calls unwind. Don't go back up the table and fill in the OUTPUT column as this will not make it clear enough when the output occurred.

Tracing a recursive function

A recursive function has a statement after the recursive call to itself: the `RETURN` statement. Again we show what happens when the recursive calls unwind by filling in more rows in the trace table. Let's consider the factorial function again.

```
FUNCTION Factorial(n : INTEGER) RETURNS INTEGER
   IF n = 0
     THEN
       Result ← 1
     ELSE
```

```
        Result ← n * Factorial(n – 1)
    ENDIF
    RETURN Result
ENDFUNCTION
```

Call number	Procedure call	n = 0	Result	Return value
1	Factorial(4)	FALSE	4 * Factorial(3)	
2	Factorial(3)	FALSE	3 * Factorial(2)	
3	Factorial(2)	FALSE	2 * Factorial(1)	
4	Factorial(1)	FALSE	1 * Factorial(0)	
5	Factorial(0)	TRUE	1	1
(4)	Factorial(1)	FALSE	1 * 1	1
(3)	Factorial(2)	FALSE	2 * 1	2
(2)	Factorial(3)	FALSE	3 * 2	6
(1)	Factorial(4)	FALSE	4 * 6	24

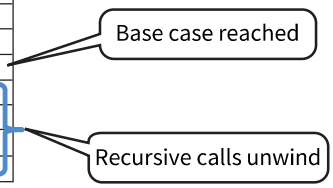


Figure 24.05 Trace table for Trace table for `CALL Factorial(4)`

Another way to illustrate how the function calls unwind is by framing each call with a box (see Figure 24.06). When the innermost box is completed the result is fed to the next one out. And so on until the outermost box has been completed.

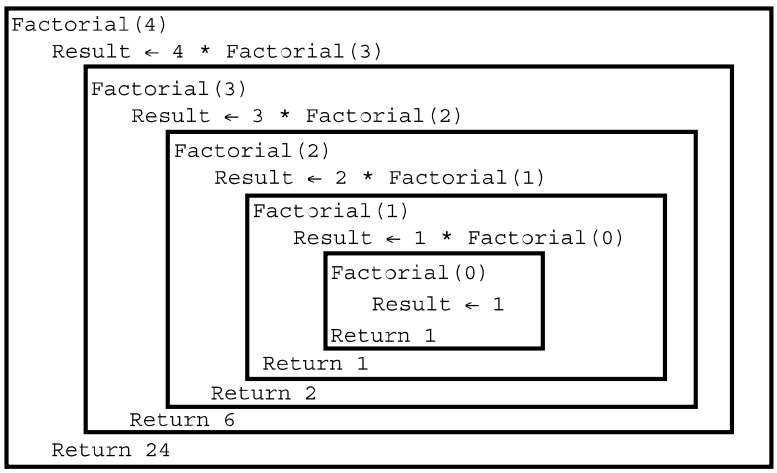


Figure 24.06 Diagrammatic view of recursive calls of `Factorial`

TASK 24.02

Consider the following recursive algorithm:

```
PROCEDURE X(n : INTEGER)
    IF (n = 0) OR (n = 1)
      THEN
        OUTPUT n
      ELSE
        CALL X(n DIV 2)
        OUTPUT(n MOD 2)
    ENDIF
ENDPROCEDURE
```

Dry-run the procedure call `X(19)` by completing a trace table. What is the purpose of this algorithm?

24.04 Running a recursive subroutine

Recursive subroutines can only be executed if the compiler produces object code that uses a stack to push return addresses and local variables when calling a subroutine repeatedly.

WORKED EXAMPLE 24.03

Running the factorial function

Consider the following program, written in pseudocode:

```
010  PROGRAM
020
030  FUNCTION Factorial(n : INTEGER) RETURNS INTEGER
040      IF n = 0
050        THEN
060          Result ← 1
070        ELSE
080          Result ← n * Factorial(n - 1)
090      ENDIF
100      RETURN Result
110  ENDFUNCTION
120
130  // main program
140
150  DECLARE Answer : INTEGER
160  Answer ← Factorial(3)
170  OUTPUT Answer
180
190  ENDPROGRAM
```

The first program statement to be executed is line 160. The actual parameter `n` has the value 3. The function call causes the return address to be put on the stack, as shown in Figure 24.07. Program execution jumps to line 30.

When line 80 is reached, the function call causes the return address to be stored on the stack, together with the current contents of the local variables. The locations used to store these values are referred to as a stack frame (represented by the blue borders in Figure 24.07). Each recursive call will add another stack frame to the stack until the base case is reached.

When the base case is reached, the result of the function call `Factorial(0)` is returned by pushing it onto the stack. The result is popped off the stack by the previous call of the function. With each return from a function call, the corresponding stack frame is taken off and the values of the local variables are restored. Eventually, control is returned to line 160 with the result of the function call on the top of the stack. The value of `Answer` is output in line 170.

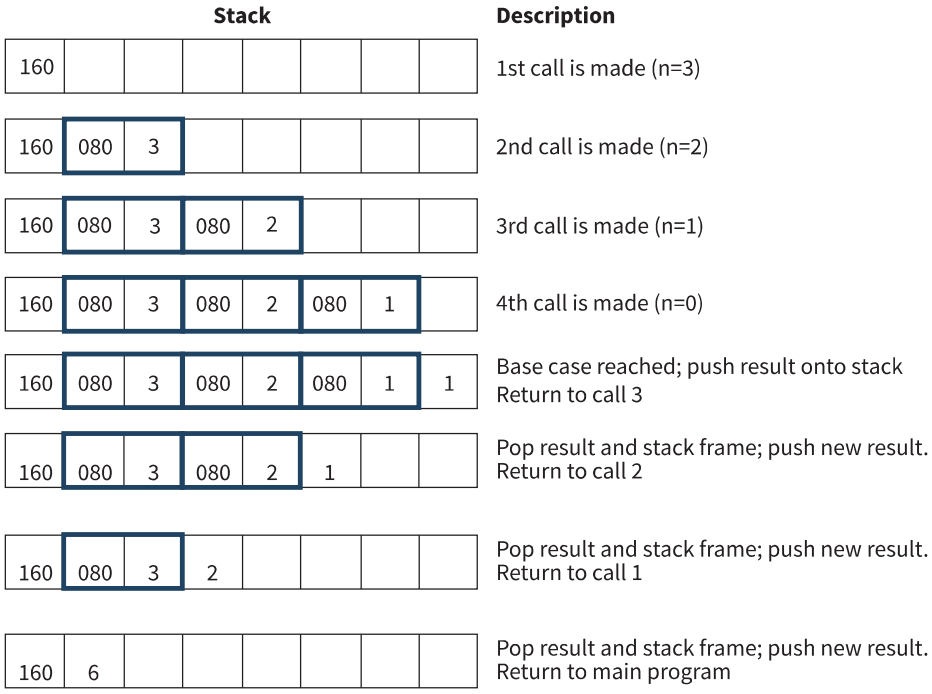

Stack
Description
160
1st call is made (n=3)
160 080 3
2nd call is made (n=2)
160 080 3 080 2
3rd call is made (n=1)
160 080 3 080 2 080 1
4th call is made (n=0)
160 080 3 080 2 080 1 1
Base case reached; push result onto stack
Return to call 3
160 080 3 080 2 1
Pop result and stack frame; push new result.
Return to call 2
160 080 3 2
Pop result and stack frame; push new result.
Return to call 1
160 6
Pop result and stack frame; push new result.
Return to main program

Figure 24.07 Stack contents during recursive calls of `Factorial`

TASK 24.03

Use your program code from Task 24.01 and add the main program as shown in Worked Example 24.03.

Amend your code in the following ways (line numbers are relative to the pseudocode in Worked Example 24.03):

- Add a global integer variable `CallNumber`
- Initialise `CallNumber` to zero (line `155`).
- Increment `CallNumber` (line `35`).
- Add a statement to output the values of `CallNumber` and `n` (line `36`).
- Add a statement to output the value of `Result` (line `95`).

Run the program and study the output.

24.05 Benefits and drawbacks of recursion

Recursion is an important technique in different programming paradigms (See Chapter 29, Section 29.08). When designing a solution to a mathematical problem that is recursive by nature, the easiest way to write a solution is to implement the recursive definition. Some optimising compilers will change a recursive subroutine to an iterative one when producing object code.

Recursive solutions are often more elegant and use less program code than iterative solutions. However, repeated recursive calls can carry large amounts of memory usage and processor time (see Section 24.04). For example, the procedure call `CountDownFrom(100)` will require 100 stack frames before it completes.

Reflection Point:

How can you tell from a function or procedure definition whether or not it is recursive?

Summary

- A recursive subroutine is defined in terms of itself.
- A recursive subroutine must have a base case and a general case.
- A recursive subroutine must reach the base case after a finite number of calls to itself.
- Each time a subroutine is called, a stack frame is pushed onto the stack.
- A stack frame consists of the return address and the values of the local variables.
- When a subroutine completes, the corresponding stack frame is popped off the stack.

Exam-style Questions

1 **a** Outline the differences between iteration and recursion. [2]

b Give **one** advantage and **one** disadvantage of using recursive subroutines. [2]

2 The following is a recursively defined function which calculates the result of $\text{Base}^{\text{Exponent}}$. For example, 2^3 is 8.

```
FUNCTION Power(Base: INTEGER, Exponent : INTEGER) RETURNS INTEGER
    IF Exponent = 0
      THEN
        Result ← 1
      ELSE
        Result ← Base * Power(Base, Exponent – 1)
    ENDIF
    RETURN Result
ENDFUNCTION
```

a What is meant by 'recursively defined'? [1]

b Trace the execution of the function call `Power(2,4)` showing for each re-entry into the `Power` function, the values passed to the function and the results returned. [6]

c Explain the role of the stack in the execution of the Power function. [3]

d Write a pseudocode non-recursive (iterative) version of the Power function. [6]

e **i** Give **one** reason why a non-recursive Power function may be preferred to a recursive one. [1]

ii Give **one** reason why a recursive Power function may be preferred to a non-recursive one. [1]

3 The following is a recursively defined function which calculates the *n*th integer in the sequence of Fibonacci numbers.

```
01  FUNCTION Fibonacci(n : INTEGER) RETURNS INTEGER
02      IF (n = 0) OR (n = 1)
03        THEN
04          Result ← 1
05        ELSE
06          Result ← Fibonacci(n – 1) + Fibonacci(n – 2)
07      ENDIF
08      RETURN Result
09  ENDFUNCTION
```

a **i** Which line is the base case? [1]

ii Which line is the general case? [1]

b Dry-run the function call `Fibonacci(4)`. [7]

Chapter 25:
Programming paradigms

Learning objectives

By the end of this chapter you should be able to:

- show understanding of what is meant by a programming paradigm
- show understanding of the characteristics of a number of programming paradigms:
  - low-level, imperative, object-oriented and declarative.

25.01 Programming paradigms

A **programming paradigm** is a fundamental style of programming. Each paradigm will support a different way of thinking and problem solving. Paradigms are supported by programming language features. Some programming languages support more than one paradigm. There are many different paradigms, and some overlap. Here are just a few different paradigms.

Low-level programming paradigm

The features of low-level programming languages give us the ability to manipulate the contents of memory addresses and registers directly and exploit the architecture of a given processor. We solve problems in a very different way when we use the low-level programming paradigm than if we use a high-level paradigm. See Chapter 6 and Chapter 28 for low-level programming examples. Note that each different type of processor has its own programming language. There are 'families' of processors that are designed with similar architectures and therefore use similar programming languages. For example, the Intel processor family (present in many PC-type computers) uses the x86 instruction set.

Imperative programming paradigm

Imperative programming involves writing a program as a sequence of explicit steps that are executed by the processor. Most of the programs in this book use imperative programming (Chapters 12 to 15 and Chapters 23, 24 and 26). An imperative program tells the computer *how* to get a desired result, in contrast to declarative programming where a program describes *what* the desired result should be. Note that the procedural programming paradigm belongs to the imperative programming paradigm. There are many imperative programming languages, Pascal, C and Basic to name just a few.

Object-oriented programming paradigm

The object-oriented paradigm is based on objects interacting with one another. These objects are data structures with associated methods (see Chapter 27). Many programming languages that were originally imperative have been developed further to support the object-oriented paradigm. Examples include Pascal (under the name Delphi or Object Pascal) and Visual Basic (the .NET version being the first fully object-oriented version). Newer languages, such as Python and Java, were designed to be object-oriented from the beginning.

Declarative programming paradigm

Programming languages such as Pascal, VB and Python are referred to as 'imperative programming languages' because the programmer writes sequences of statements that reflect how to solve the problem. When a programmer uses a declarative programming language, the programmer writes down (in the language of logic) a declarative specification that describes the situation of interest: *what* the problem is. The programmer doesn't tell the computer what to do. To get information, the programmer poses a query (sets a goal). It's up to the logic programming system to work out how to get the answer.

Declarative programs are expressed as formal logic and computations are deductions from the formal logic statements (see Chapter 29). Declarative programming languages include SQL (see Chapter 11) and Prolog (Chapter 29).

Reflection Point:

Why are there different paradigms?

After you have studied Chapters 26 to 29 do Tasks 25.01 and 25.02

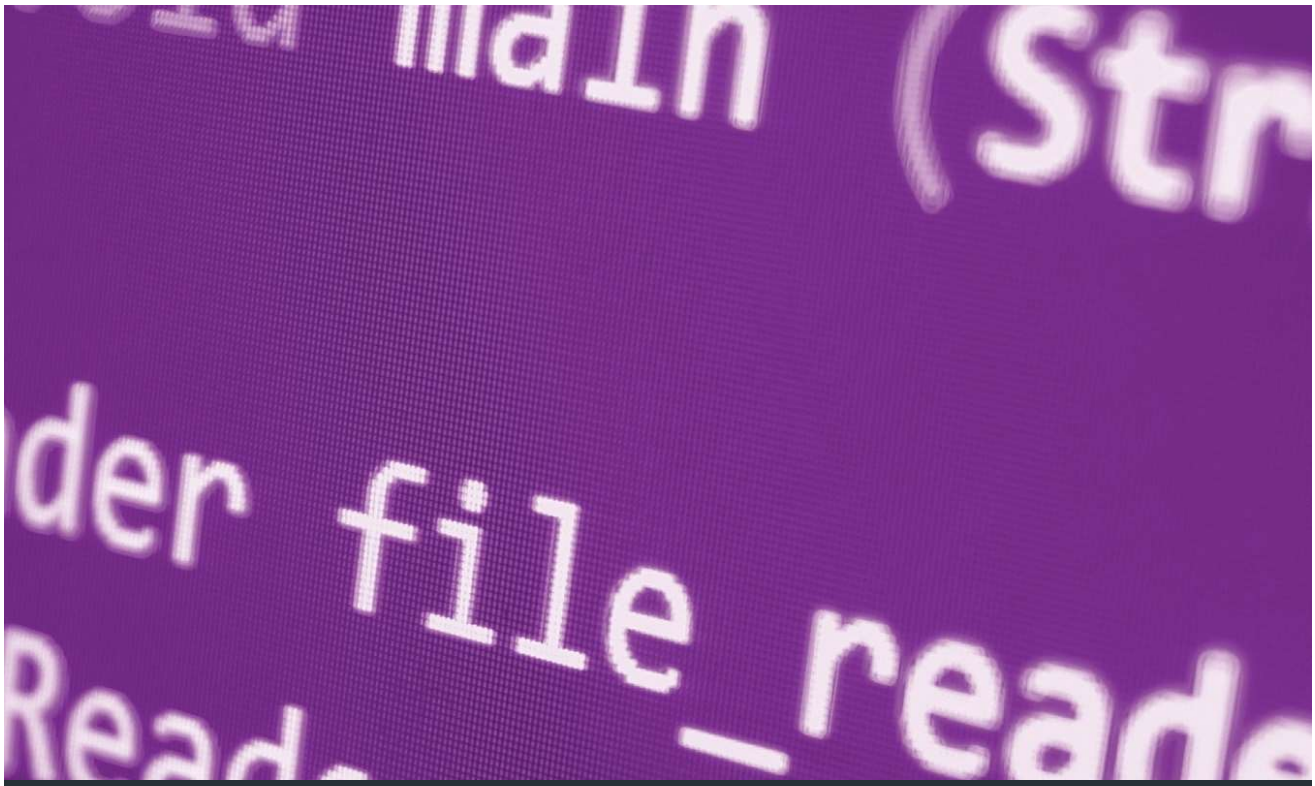
TASK 25.01

Draw a line between a statement on the left and its matching programming paradigm on the right.

Commands available depend on the type of processor	Declarative
Information hiding is used to protect internal properties of an object	
Statements operate on data	Imperative
Answer a question via a search for a solution	
Data is combined with procedures	Low-level
First do this and then do that	
The basic concept is a relation	Object-oriented

TASK 25.02

For each of the four programming paradigms in this chapter, give one programming statement example that is characteristic for this paradigm. State the programming language you used for each example.



Chapter 26:
File processing and exception handling

Learning objectives

By the end of this chapter you should be able to:

- write code to perform file-processing operations:
  - open a file (in read, write or append mode)
  - read a record from a file
  - write a record to a file
  - on serial, sequential and random files
- show understanding of an exception and the importance of exception handling
- show understanding of when it is appropriate to use exception handling
- write code to use exception handling.

26.01 Records

Records are user-defined types (discussed in Chapter 16, Section 16.01). See also Section 13.02.

WORKED EXAMPLE 26.01

Using records

A car manufacturer and seller wants to store details about cars. These details can be stored in a record structure:

```
TYPE CarRecord
    DECLARE VehicleID           : STRING  // unique identifier and record key
    DECLARE Registration        : STRING
    DECLARE DateOfRegistration  : DATE
    DECLARE EngineSize          : INTEGER
    DECLARE PurchasePrice       : CURRENCY
ENDTYPE
```

To declare a variable of that type we write:

```
DECLARE ThisCar : CarRecord
```

Note that we can declare arrays of records. If we want to store the details of 100 cars, we declare an array of type `CarRecord`

```
DECLARE Car : ARRAY[1:100] OF CarRecord
```

Python

Python does not have a record type. However, we can use a class definition with only a constructor to assign initial values. (See Chapter 27 for more about classes).
The pseudocode example of a car record described in Worked Example 26.01 can be programmed as follows:

```
class CarRecord:        # declaring a class without other methods
  def __init__(self):  # constructor
     self.VehicleID = ""
     self.Registration = ""
     self.DateOfRegistration = None
     self.EngineSize = 0
     self.PurchasePrice = 0.00
ThisCar = CarRecord() # instantiates a car record
ThisCar.EngineSize = 2500 # assign a value to a field
Car = [CarRecord() for i in range(100)] # make a list of 100 car records
Car[1].EngineSize = 2500 # assign value to a field of the 2nd car in list
```

VB.NET

```
Structure CarRecord
   Dim VehicleID As String
   Dim Registration As String
   Dim DateOfRegistration As Date
   Dim EngineSize As Integer
   Dim PurchasePrice As Decimal
End Structure
Dim ThisCar As CarRecord    ' declare a variable of CarRecord type
Dim Car(100) As CarRecord   ' declare an array of CarRecord type
ThisCar.EngineSize = 2500 ' assign value to a field
Car(2).EngineSize = 2500  ' assign value to a field of 2nd car in array
```

Java

Java does not have a record type. However, we can use a data structure. A class definition without methods is a data structure that can be used like a record. (See Chapter 27 for more

about classes).

The pseudocode example of a car record described in Worked Example 26.01 can be programmed as follows:

```
class CarRecord
{
   String vehicleID;
   String registration;
   String dateOfRegistration;
   int engineSize;
   double purchasePrice;
  public CarRecord()   // declare a constructor without other methods
   {
      vehicleID = "XX";
      registration = "";
      dateOfRegistration = "01/01/2010";
      engineSize = 0;
      purchasePrice = 0.00;
   }
}

CarRecord thisCar = new CarRecord(); // instantiates a car record
thisCar.engineSize = 2500; // assign a value to a field

CarRecord[] car = new CarRecord[100]; // declare an array of car record type

car[2] = new CarRecord(); // instantiate a car
car[2].engineSize = 2500; // assign a value to a field of 2nd car in array
```

26.02 File processing

In Chapter 13 (Section 13.06) we used text files to store and read lines of text. Text files only allow us to write strings in a serial or sequential manner. We can append strings to the end of the file.

When we want to store records in a file, we create a binary file (see Chapter 16, Section 16.02). We can store records serially or sequentially. We can also store records using direct access to a **random file**. Table 26.01 lists the operations we use for processing files.

Structured English	Pseudocode
Create a file and open it for writing	`OPENFILE <filename> FOR WRITE`
Open a file in append mode	`OPENFILE <filename> FOR APPEND`
Open a file for reading	`OPENFILE <filename> FOR READ`
Open a file for random access	`OPENFILE <filename> FOR RANDOM`
Close a file	`CLOSEFILE <filename>`
Write a record to a file	`PUTRECORD <filename>, <identifier>`
Read a record from a file	`GETRECORD <filename>, <identifier>`
Move to a specific disk address within the file	`SEEK <filename>, <address>`
Test for end of file	`EOF(<filename>)`

Table 26.01 Operations for file processing

Sequential file processing

If we have an array of records, we may want to store the records on disk before the program quits, so that we don't lose the data. We can open a binary file and write one record after another to the file. We can then read the records back into the array when the program is run again.

WORKED EXAMPLE 26.02

Processing records in a sequential file

Table 26.02 shows the pseudocode for storing the car records from Worked Example 26.01 in a sequential file and accessing them.

Saving contents of array	Restoring contents of array

`OPENFILE CarFile FOR WRITE`
`FOR i ← 1 TO MaxRecords`
`  PUTRECORD CarFile, Car[i]`
`NEXT i`
`CLOSEFILE CarFile`
	`OPENFILE CarFile FOR READ`
	`FOR i ← 1 TO MaxRecords`
	`  GETRECORD CarFile, Car[i]`
	`NEXT i`
	`CLOSEFILE CarFile`

Table 26.02 Pseudocode for processing records

Processing records sequentially in Python, VB.NET and Java

Python

```
import pickle # this library is required to create binary files
Car = [CarRecord() for i in range(100)]
CarFile = open('CarFile.DAT', 'wb') # open file for binary write
for i in range(100):  # loop for each array element
   pickle.dump(Car[i], CarFile) # write a whole record to the binary file
CarFile.close() # close file
CarFile = open('CarFile.DAT', 'rb') # open file for binary read
Car = [] # start with empty list
while True: # check for end of file
   Car.append(pickle.load(CarFile)) # append record from file to end of list
```

```
CarFile.close()
```

VB.NET

```
Option Explicit On
Imports System.IO

Dim CarFileWriter As BinaryWriter
Dim CarFileReader As BinaryReader
Dim CarFile As FileStream
Dim Car(100) As CarRecord ' declare an array of CarRecord type
Dim i As Integer

'link file to the filename
CarFile = New FileStream("CarFile.DAT", FileMode.Create)
' create a new file and open it for writing
CarFileWriter = New BinaryWriter(CarFile)
For i = 1 To 100 ' loop for each array element
   CarFileWriter.Write(Car(i).VehicleID) ' write a field to the binary file
   CarFileWriter.Write(Car(i).Registration)
   CarFileWriter.Write(Car(i).DateOfRegistration)
   CarFileWriter.Write(Car(i).EngineSize)
   CarFileWriter.Write(Car(i).PurchasePrice)
Next

CarFileWriter.Close()   ' close file channel
CarFile.Close()

'link file to the filename
CarFile = New FileStream("CarFile.DAT", FileMode.Open)
' create a new file and open it for reading
CarFileReader = New BinaryReader(CarFile)
i = 0

' loop until end of binary file reached
i = 0
Do While CarFile.Position < CarFile.Length
   ' read fields from the binary file
   Car(i).VehicleID = CarFileReader.ReadString()
   Car(i).Registration = CarFileReader.ReadString()
   Car(i).DateOfRegistration = CarFileReader.ReadString()
   Car(i).EngineSize = CarFileReader.ReadInt32()
   Car(i).PurchasePrice = CarFileReader.ReadDecimal()
   i = i + 1
Loop

CarFileReader.Close()   'close file channel

CarFile.Close()
```

Java

writing records to a file:

```
import java.io.DataOutputStream;
import java.io.FileOutputStream;
import java.io.EOFException;
import java.io.FileInputStream;
import java.io.FileOutputStream;
   try
   {  // set up file stream and link to file name
      FileOutputStream fos = new FileOutputStream("CarFile.DAT");
      // link file stream to data stream
      DataOutputStream dos = new DataOutputStream(fos);
      for (int i = 1; i < 100; i++) // loop for each array element
      {
         dos.writeUTF(car[i].vehicleID); // write a field to the file
         dos.writeUTF(car[i].registration);
         dos.writeInt(car[i].engineSize);
         dos.writeUTF(car[i].dateOfRegistration);
         dos.writeDouble(car[i].purchasePrice);
      }
      dos.close(); // close data stream
   }
   catch (Exception x)
   {
      System.out.println("IO error");
   }
```

reading records back into array:

```
   CarRecord[] car = new CarRecord[100];
   try
   {  // set up file stream and link to file name
      FileInputStream fis = new FileInputStream("CarFile.DAT");
      // link file stream to data stream
      DataInputStream dis = new DataInputStream(fis);
      int i = 1;
      while (true) // loop for each array element
      {
         thisCar = new CarRecord();
         thisCar.vehicleID = dis.readUTF(); // read fields from the file
         thisCar.registration = dis.readUTF();
         thisCar.engineSize = dis.readInt();
         thisCar.dateOfRegistration = dis.readUTF();
         thisCar.purchasePrice = dis.readDouble();
         car[i] = thisCar; // assign record to next array element
         i += 1;
      }
   }
   catch (EOFException x)
   {
      System.out.println("End of File reached");
   }
   catch (Exception x)
   {
      System.out.println(x);  // output error message
   }
```

Alternative method using Date data type (which is an object in Java):

write to file

```
import java.io.*;
import java.util.Date;

class CarRecord implements java.io.Serializable // to allow writing and reading
                                                 // objects
{
   String vehicleID;
   String registration;
   Date dateOfRegistration;
   int engineSize;
   double purchasePrice;


   public CarRecord() // declare constructor
   {
      vehicleID = "XX";
      registration = "";
      dateOfRegistration = new Date();
      engineSize = 0;
      purchasePrice = 0.00;
   }
}


class Program
{
public static void main(String[] args)
{
   CarRecord[] car = new CarRecord[100];
   CarRecord thisCar;

   try
   {  //Write array to file.

      FileOutputStream fos = new FileOutputStream("cars.ser");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      for (int i = 1; i < 100; i++)  // loop for each array element
      {
         oos.writeObject(car[i].vehicleID); // write a field to the file
         oos.writeObject(car[i].registration);
         oos.writeObject(car[i].engineSize);
         oos.writeObject(car[i].dateOfRegistration);
```

```
            oos.writeObject(car[i].purchasePrice);
         }
         oos.close();
      }
      catch (EOFException x)
      {
         System.out.println("End of File reached");
      }
         catch (Exception x)
      {
         System.out.println(x);
      }
}
}
```

reading file into array:

```
CarRecord[] car = new CarRecord[100];
try
{
   //Read array from file.
   FileInputStream fis = new FileInputStream("cars.ser");
   ObjectInputStream ois = new ObjectInputStream(fis);
   int i = 1;
   while (true)
   {
      car[i].vehicleID = (String) ois.readObject(); // read the first item from the
                                          // file to the field
      car[i].registration= (String) ois.readObject();
      car[i].engineSize = (int) ois.readObject();
      car[i].dateOfRegistration = (Date) ois.readObject();
      car[i].purchasePrice = (double) ois.readObject();
      i += 1;
   }
}
catch (EOFException x)
{
   System.out.println("End of File reached");
}
catch (Exception x)
{
   System.out.println(x);
}
```

TASK 26.01

1. Write a complete program to save several car records to a sequential file.
2. Write another program to read the file and display the contents on screen.

Random-access file processing

Instead of storing records in an array, we may want to store each record in a binary file as the record is created. We can then update the record in situ (read it, change it and save it back in the same place). Note that this only works for fixed-length records. We can use a hashing function to calculate an address from the record key and store the record at the calculated address in the file (this is similar to using a hash table, see Chapter 23, Section 23.11). Just as with a hash table, collisions may occur and records need to be stored in the next free record space.

WORKED EXAMPLE 26.03

Processing records in a random-access file

Table 26.03 shows the pseudocode for storing a car record from Worked Example 26.01 in a random-access file and accessing it.

Saving a record	Retrieving a record

`OPENFILE CarFile FOR RANDOM`
`Address ← Hash(ThisCar.VehicleID)`
`SEEK CarFile, Address`
`PUTRECORD CarFile, ThisCar`
`CLOSEFILE CarFile`
	`OPENFILE CarFile FOR RANDOM`
	`Address ← Hash(ThisCar.VehicleID)`
	`SEEK CarFile, Address`
	`GETRECORD CarFile, ThisCar`
	`CLOSEFILE CarFile`

Table 26.03 Pseudocode for random-access file operations

`SEEK` moves a pointer to the given record address. The `PUTRECORD` and `GETRECORD` commands access the record at that address. After the command has been executed the pointer points to the next record in the file.

Processing random-access records in Python, VB.NET and Java

Python

```
import pickle  # this library is required to create binary files
ThisCar = CarRecord()
CarFile = open('CarFile.DAT','rb+')  # open file for binary read and write
Address = hash(ThisCar.VehicleID)
CarFile.seek(Address)
pickle.dump(ThisCar, CarFile)  # write a whole record to the binary file
CarFile.close()  # close file
```

to find a record from a given VehicleID:

```
CarFile = open('CarFile.DAT','rb')  # open file for binary read
Address = hash(VehicleID)
CarFile.seek(Address)
ThisCar = pickle.load(CarFile)  # load record from file
CarFile.close()
```

In Python, the hash function needs to allow for the record size in bytes. For example, if the record size is 58 bytes, then the second record slot starts at position 59. The nth record slot starts at position $(n - 1) \times 58 + 1$.

VB.NET

```
Dim CarFileWriter As BinaryWriter
Dim CarFileReader As BinaryReader
Dim CarFile As FileStream
Dim ThisCar, MyCar As CarRecord
' link the file to the filename
CarFile = New FileStream("CarFile.DAT", FileMode.Open)
' create a new file and open it for writing
CarFileWriter = New BinaryWriter(CarFile)
' get starting address for record
CarFile.Position = Hash(ThisCar.VehicleID)
' write fields to the binary file
CarFileWriter.Write(ThisCar.VehicleID)
CarFileWriter.Write(ThisCar.Registration)
CarFileWriter.Write(ThisCar.DateOfRegistration)
CarFileWriter.Write(ThisCar.EngineSize)
CarFileWriter.Write(ThisCar.PurchasePrice)
CarFileWriter.Close() 'close file channel
CarFile.Close()
```

to find a record from a given VehicleID:

```
CarFile = New FileStream("CarFile.DAT", FileMode.Open)
CarFileReader = New BinaryReader(CarFile)
' get starting address for record
CarFile.Position = Hash(VehicleID)
' read fields from the binary file
MyCar.VehicleID = CarFileReader.ReadString()
MyCar.Registration = CarFileReader.ReadString()
MyCar.DateOfRegistration = CarFileReader.ReadString()
```

```
MyCar.EngineSize = CarFileReader.ReadInt32()
MyCar.PurchasePrice = CarFileReader.ReadDecimal()
CarFileReader.Close() 'close file channel
CarFile.Close()
```

In VB.NET, the hash function needs to allow for the record size in bytes. For example, if the record size is 58 bytes, then the second record slot starts at position 59. The nth record slot starts at position $(n - 1) \times 58 + 1$.

Java

```
CarRecord thisCar;
int recordSize = 50;
try // set up a file with 100 dummy records
{
      RandomAccessFile writer = new RandomAccessFile("CarFile.DAT", "rw");
      for (int i = 1; i < 100; i++) // loop for each array element
      {
      thisCar = new CarRecord();
      thisCar.vehicleID = "A" + i;
      writer.seek(i * recordSize);
      writer.writeUTF(thisCar.vehicleID);
      writer.writeUTF(thisCar.registration);
      writer.writeUTF(thisCar.dateOfRegistration);
      writer.writeInt(thisCar.engineSize);
      writer.writeDouble(thisCar.purchasePrice);
      }
      writer.close();
}
catch (IOException x)
{
}
```

to find a record from a given VehicleID:

```
try
{
      RandomAccessFile writer = new RandomAccessFile("CarFile.DAT", "rw");
      RandomAccessFile reader = new RandomAccessFile("CarFile.DAT", "r");
      reader.seek(hash(vehicleID));
      thisCar = new CarRecord();
      thisCar.vehicleID = reader.readUTF();
      thisCar.registration = reader.readUTF();
      thisCar.engineSize = reader.readInt();
      thisCar.dateOfRegistration = reader.readUTF();
      thisCar.purchasePrice = reader.readDouble();
      reader.close();
}
catch (IOException x)
{
}
```

TASK 26.02

Write a complete program to save several car records to a random-access file. Write another program to find a record in the random-access file using the record key. Display the record data on screen.

26.03 Exception handling

Run-time errors can occur for many reasons. Some examples are division by zero, invalid array index or trying to open a non-existent file. Run-time errors are called 'exceptions'. They can be handled (resolved) with an error subroutine (known as an 'exception handler'), rather than let the program crash.

Using pseudocode, the error-handling structure is:

```
TRY
   <statementsA>
EXCEPT
   <statementsB>
ENDTRY
```

Any run-time error that occurs during the execution of `<statementsA>` is caught and handled by executing `<statementsB>`. There can be more than one `EXCEPT` block, each handling a different type of exception. Sometimes a `FINALLY` block follows the exception handlers. The statements in this block will be executed regardless of whether there was an exception or not.

VB.NET is designed to treat exceptions as abnormal and unpredictable erroneous situations. Python is designed to use exception handling as flow-control structures. You may find you need to include exception handling in the code for Worked Example 26.02. Otherwise the end of file is encountered and the program crashes.

Python distinguishes between different types of exception, such as:

- `IOError`: for example, a file cannot be opened
- `ImportError`: Python cannot find the module
- `ValueError`: an argument has an inappropriate value
- `KeyboardInterrupt`: the user presses Ctrl+C or Ctrl+Del
- `EOFError`: a file-read meets an end-of-file condition
- `ZeroDivisionError`: a division by zero has been attempted

Java distinguishes between different types of exception, such as:

- `IOException`: for example, a file cannot be opened
- `ArithmeticException`: an arithmetic error occurred, such as division by zero
- `ArrayIndexOutOfBoundsException`: the program tries to access an array element outside the boundary

WORKED EXAMPLE 26.04

Here is a simple example of exception handling. Asking the user to key in an integer could result in a non-integer input. This should not crash the program.

Python

```
NumberString = input("Enter an integer: ")
try:
   n = int(NumberString)
   print(n)
except:
   print("This was not an integer")
```

VB.NET

```
Dim NumberString As String
Dim n As Integer
Console.WriteLine("Enter an integer")
NumberString = Console.ReadLine()
Try
```

```
   n = Int(NumberString)
   Console.WriteLine(n)
Catch
   Console.WriteLine("This was not an integer")
End Try
```

Java

```
import java.util.Scanner;
{
   public static void main(String[] args)
   {
      Scanner console = new Scanner(System.in);
      try
      {
         System.out.print("Enter an integer: ");
         int n = console.nextInt();
         System.out.println(n);
      }
      catch(Exception e) // catches any exception
      {
         System.out.println("This was not an integer");
      }
   }
}
```

TASK 26.03

Add exception-handling code to your programs for Task 26.01 or Task 26.02. Test your code handles exceptions without the program crashing.

Reflection Point:

Can you explain the difference between serial, sequential and random access files? Give an example of when each is appropriate.

Summary

- Records are user-defined types.
- Records can be stored in files in a serial, sequential or random (direct access) manner.
- Exception handling is advisable to avoid program crashes due to run-time errors.

Exam-style Questions

1 A company stores details about their customers in a binary file of records.

- The key field of a customer record is the customer ID (a number between 100001 and 999999).
- The name of the customer is stored in a 30-character field.
- The customer's telephone number is stored in a 14-character field.
- The total value of orders so far is stored in a currency (decimal) field.

a **i** Write the declaration statement for the record data type `CustomerRecord` required to store the data. Write program code. [6]

ii Write the declaration statement for an array `CustomerData[0 : 999]` to store customer records.

b The array `CustomerData` is to be used as a hash table to store customer records. The function `Hash` is used to calculate the address where a record is to be stored. [2]

```
FUNCTION Hash(CustomerID : INTEGER) RETURNS INTEGER
    Address ← CustomerID MOD 1000
    RETURN Address
ENDFUNCTION
```

i Write program code to implement the function `Hash`. [3]

ii Write a procedure `AddRecord(Customer : CustomerRecord)` to add a customer record to the hash table `CustomerData`. Your solution should handle collisions by using the next available slot in the hash table. [7]

iii Write a function `FindRecord(CustomerID : INTEGER)` that returns the index of the hash table slot where the record for the customer with `CustomerID` is stored. [7]

c Before the program stops, the hash table records must be stored in a sequential file, so that the records can be restored to the array when the program is re-entered.

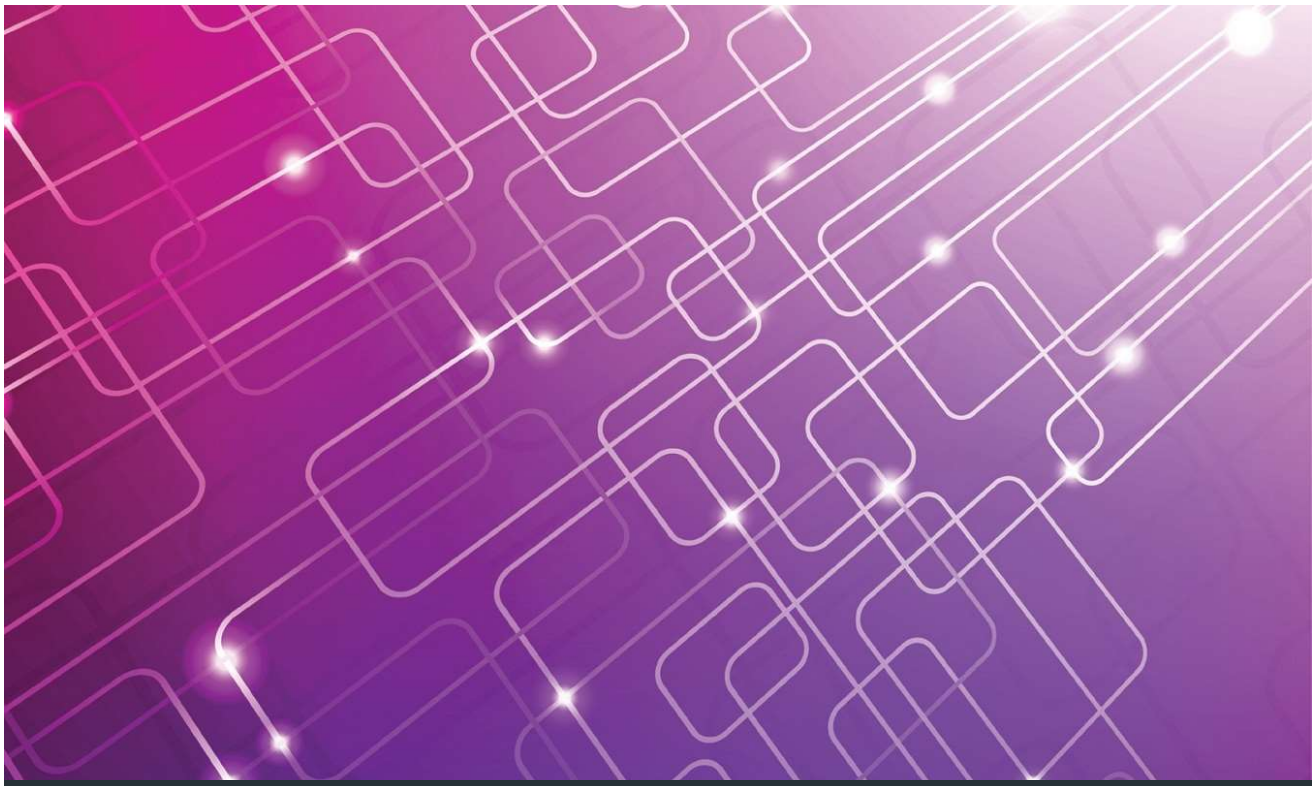
Write program code to store the records of the array CustomerData sequentially into a binary file CustomerData.DAT [6]

d Instead of using a hash table, the company decide they want to store customer records in a direct-access binary file.

Explain what changes need to be made to your program to do this. [6]

2 A program allows a user to enter a filename for accessing a data file. If the user types in a filename that does not exist, the program crashes. Write program code that includes exception handling to replace the following pseudocode: [5]

```
OUTPUT "Which file do you want to use? "
INPUT FileName
OPENFILE FileName FOR RANDOM
```



Chapter 27:
Object-Oriented Programming (OOP)

Learning objectives

By the end of this chapter you should be able to:

- show understanding of the terminology associated with OOP: objects, properties, methods, classes, inheritance, polymorphism, containment/aggregation, encapsulation, getters, setters and instances
- show understanding of how to solve a problem by designing appropriate classes
- show understanding of and write code that demonstrates the use of OOP.

27.01 Concept of OOP

Chapters 14 and 26 covered programming using the procedural aspect of our programming languages. Procedural programming groups related programming statements into subroutines. Related data items are grouped together into record data structures. To use a record variable, we first define a record type. Then we declare variables of that record type.

Example of using a record

A car manufacturer and seller wants to store details about cars. These details can be stored in a record structure (see Chapter 16, Section 16.01 and Chapter 26, Section 26.02):

```
TYPE CarRecord
    DECLARE VehicleID            : STRING
    DECLARE Registration         : STRING
    DECLARE DateOfRegistration   : DATE
    DECLARE EngineSize           : INTEGER
    DECLARE PurchasePrice        : CURRENCY
ENDTYPE
```

We can write program code to access and assign values to the fields of this record. For example:

```
PROCEDURE UpdateRegistration(BYREF ThisCar : CarRecord, BYVALUE NewRegistration)
    ThisCar.Registration ← NewRegistratio
ENDPROCEDURE
```

We can call this procedure from anywhere in our program. This seems a well-regulated way of operating on the data record. However, we can also access the record fields directly from anywhere within the scope of `ThisCar`:

```
ThisCar.EngineSize ← 2500
```

Classes in OOP

OOP goes one step further and groups together the data structure and the subroutines that operate on the data items in this data structure. Such a group is called an **object**. The data of an object are called **attributes** and the subroutines acting on the attributes are called **methods**. The idea behind OOP is that attributes can only be accessed through methods. The direct path to the data is unavailable. Attributes are referred to as ‘private’. The methods to access the data are made available to programmers, so these are ‘public’. The feature of data being combined with the subroutines acting on this data is known as **encapsulation**. To use an object, we first define an object type. An object type is called a **class**.

Classes are templates for objects. When a class type has been defined it can be used to create one or more objects of this class type. Therefore, an object is an instance of a class.

The first stage of writing an object-oriented program to solve a problem is to design the classes. This is part of object-oriented design. From this design, a program can be written using an object-oriented programming (OOP) language.

The programming languages the syllabus prescribes can be used for OOP: Python 3, VB.NET and Java.

Advantages of OOP over procedural languages

The advantage of OOP is that it produces robust, more reliable code. The attributes can only be manipulated using methods provided by the class definition. This means the attributes are protected from accidental changes. Classes provided in module libraries are thoroughly tested. If you use tried and tested building blocks to construct your program, you are less likely to introduce bugs than when you write code from scratch.

27.02 Designing classes and objects

When designing a class, we need to think about the attributes we want to store. We also need to think about the methods we need to access the data and assign values to the data of an object. A data type is a blueprint when declaring a variable of that data type. A class definition is a blueprint when declaring an object of that class. Creating a new object is known as 'instantiation'.

Any data that is held about an object must be accessible, otherwise there is no point in storing it. We therefore need methods to access each one of these attributes. These methods are usually referred to as **getters**. They get an attribute of the object.

When we first set up an object of a particular class, we use a constructor. A **constructor** instantiates the object and assigns initial values to the attributes.

Any attributes that might be updated after instantiation will need subroutines to update their values. These are referred to as **setters**. Some attributes get set only at instantiation. These don't need setters. This makes an object more robust, because you cannot change attributes that were not designed to be changed.

Python and VB.NET allow properties to be declared. A **property** combines the attribute with its associated setter and/or getter (see Figures 27.03 and 27.05).

WORKED EXAMPLE 27.01

Creating a class

Consider the car data from Section 27.01.

When a car is manufactured it is given a unique vehicle ID that will remain the same throughout the car's existence. The engine size of the car is fixed at the time of manufacture. The registration ID will be given to the car when the car is sold.

In our program, when a car is manufactured, we want to create a new car object. We need to instantiate it using the constructor. Any attributes that are already known at the time of instantiation can be set with the constructor. In our example, `VehicleID` and `EngineSize` can be set by the constructor. The other attributes are assigned values at the time of purchase and registration. So, we need setters for them. The identifier table for the `Car` class is shown in Table 27.01.

Identifier	Data Type	Description
`Car`	`Class`	Class identifier
`VehicleID`	`STRING`	Unique ID assigned at time of manufacture
`Registration`	`STRING`	Unique ID assigned after time of purchase
`DateOfRegistration`	`DATE`	Date of registration
`EngineSize`	`INTEGER`	Engine size assigned at time of manufacture
`PurchasePrice`	`CURRENCY`	Purchase price assigned at time of purchase
`Constructor()`		Method to create a `Car` object and set properties assigned at manufacture
`SetPurchasePrice()`		Method to assign purchase price at time of purchase
`SetRegistration()`		Method to assign registration ID
`SetDateOfRegistration()`		Method to assign date of registration
`GetVehicleID()`		Method to access vehicle ID
`GetRegistration()`		Method to access registration ID

`GetDateOfRegistration()`		Method to access date of registration
`GetEngineSize()`		Method to access engine size
`GetPurchasePrice()`		Method to access purchase price

Table 27.01 Identifier table for `Car` class

We can represent this information as a class diagram in Figure 27.01.

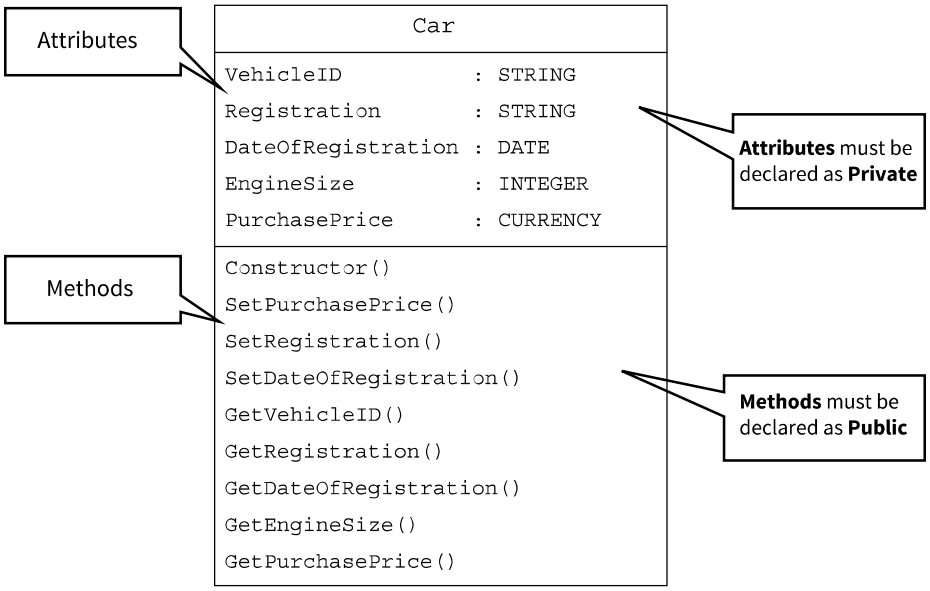


Figure 27.01 Car class diagram

27.03 Writing object-oriented code

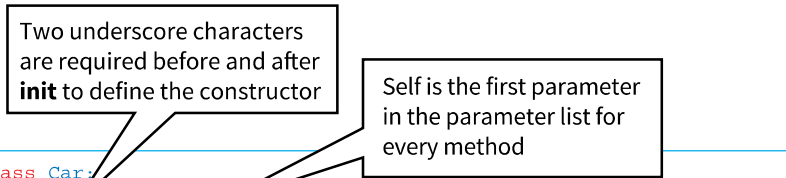
Declaring a class

Attributes should always be declared as 'Private'. This means they can only be accessed through the class methods. So that the methods can be called from the main program, they have to be declared as 'Public'. There are other modifiers (such as 'Protected'), but they are beyond the scope of this book.

The syntax for declaring classes is quite different for the different programming languages. We will look at the three chosen languages. You are expected to write programs in one of these.

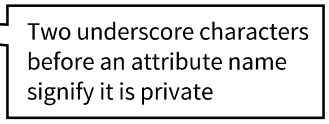
Declaring a class in Python

The code below shows how a constructor, getters and setters can be declared in Python.


Two underscore characters are required before and after **init** to define the constructor
Self is the first parameter in the parameter list for every method

Python

```
class Car:

    def __init__(self, n, e):        # constructor
        self.__VehicleID = n
        self.__Registration = ""
        self.__DateOfRegistration = None
        self.__EngineSize = e
        self.__PurchasePrice = 0.00

    def SetPurchasePrice(self, p):
        self.__PurchasePrice = p
    def SetRegistration(self, r):
        self.__Registration = r
    def SetDateOfRegistration(self, d):
        self.__DateOfRegistration = d

    def GetVehicleID ID(self) :
        return(self.__VehicleID)

    def GetRegistration(self):
        return(self.__Registration)

    def GetDateOfRegistration(self):
        return(self.__DateOfRegistration)

    def GetEngineSize(self):
        return(self.__EngineSize)

    def GetPurchasePrice(self):
        return(self.__PurchasePrice)
```



Python also supports properties:

Python using properties

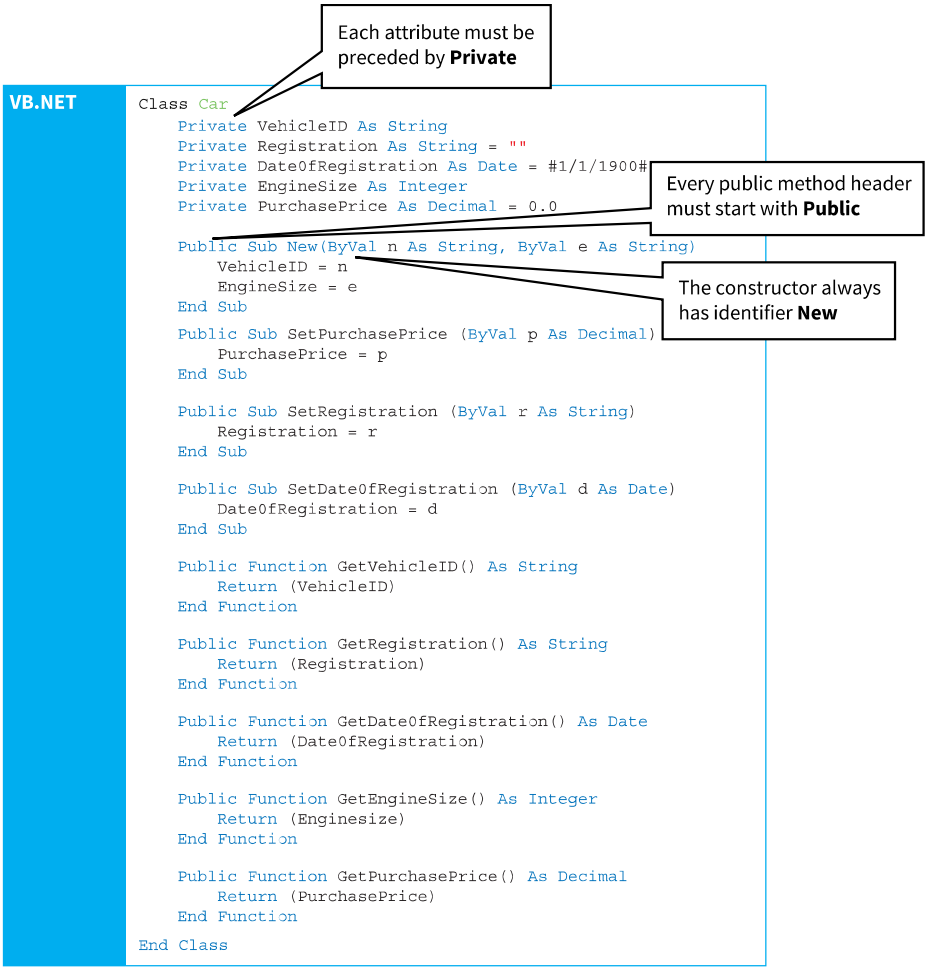
```
class Car:
    def __init__(self, n, e):    # constructor
        self.__VehicleID = n
        self.__Registration = ""
        self.__DateOfRegistration = None
        self.__EngineSize = e
        self.__PurchasePrice = 0.00

    @property
    def VehicleID(self):
        return(self.__VehicleID)

    @property
    def EngineSize(self):
        return(self.__EngineSize)

    @property
```

```
    def Registration(self):
        return(self.__Registration)

    @Registration.setter
    def Registration(self, r):
        self.__Registration = r

    @property
    def DateOfRegistration(self):
        return(self.__DateOfRegistration)

    @DataOfRegistration.setter
    def DateOfRegistration(self, d):
        self.__DateOfRegistration = d

    @property
    def PurchasePrice(self):
        return(self.__PurchasePrice)

    @PurchasePrice.setter
    def PurchasePrice(self, p):
        self.__PurchasePrice = p
```

Note that not all properties provide a setter, only where attribute values are to be changed.

Declaring a class in VB.NET

The code below shows how attributes, the constructor, getters and setters can be declared in VB.NET.


Each attribute must be preceded by **Private**
Every public method header must start with **Public**
The constructor always has identifier **New**

VB.NET

```
Class Car
    Private VehicleID As String
    Private Registration As String = ""
    Private DateOfRegistration As Date = #1/1/1900#
    Private EngineSize As Integer
    Private PurchasePrice As Decimal = 0.0

    Public Sub New(ByVal n As String, ByVal e As String)
        VehicleID = n
        EngineSize = e
    End Sub
    Public Sub SetPurchasePrice (ByVal p As Decimal)
        PurchasePrice = p
    End Sub

    Public Sub SetRegistration (ByVal r As String)
        Registration = r
    End Sub

    Public Sub SetDateOfRegistration (ByVal d As Date)
        DateOfRegistration = d
    End Sub

    Public Function GetVehicleID() As String
        Return (VehicleID)
    End Function

    Public Function GetRegistration() As String
        Return (Registration)
    End Function

    Public Function GetDateOfRegistration() As Date
        Return (DateOfRegistration)
    End Function

    Public Function GetEngineSize() As Integer
        Return (Enginesize)
    End Function

    Public Function GetPurchasePrice() As Decimal
        Return (PurchasePrice)
    End Function
End Class
```

VB.NET also supports properties:

VB using properties

```
Module Module1

Class Car
    Public Property VehicleID() As String
    Public Property Registration() As String = ""
    Public Property DateOfRegistration() As Date = #1/1/1900#
```

```
    Public Property EngineSize() As Integer
    Public Property PurchasePrice() As Decimal = 0.0

    Public Sub New(ByVal n As String, ByVal e As String)
        VehicleID = n
        EngineSize = e
    End Sub

End Class
```

Note that getters and setters are automatically available for any property.
If some properties should be read-only, getters and setters have to be declared explicitly:

```
Class Car
   Private _VehicleID As String
   Public ReadOnly Property VehicleID() As String
      Get
         Return _VehicleID
      End Get
   End Property

   Public Property Registration() As String
   Public Property DateOfRegistration() As Date

   Private _EngineSize As Integer
   Public ReadOnly Property EngineSize() As Integer
      Get
         Return _EngineSize
      End Get
   End Property

   Public Property PurchasePrice() As Decimal
   Public Sub New(ByVal n As String, ByVal e As String)
      _VehicleID = n
      _Registration = ""
      _DateOfRegistration = #1/1/1900#
      _EngineSize = e
      _PurchasePrice = 0.0
   End Sub

End Class
```

Declaring a class in Java

The code below shows how attributes, the constructor and methods are declared in Java.

Java

```
import java.util.Date;

class Car

{
    private String vehicleID;
    private String registration;
    private Date dateOfRegistration;
    private int engineSize;
    private float purchasePrice;

    public Car(String n, int e)  // constructor
    {
       vehicleID = n;
       registration = "";
       dateOfRegistration = new Date();
       engineSize = e;
       purchasePrice = 0;
    }
    public void setPurchasePrice(float p)
    {
       purchasePrice = p;
    }
    public void setRegistration(String r)
```

```
    {
       registration = r;
    }
    public void setDateOfRegistration(Date d)
    {
       dateOfRegistration = d;
    }
    public String getVehicleID()
    {
       return(vehicleID);
    }
    public String getRegistration()
    {
       return(registration);
    }
    public Date getDateOfRegistration()
    {
       return(dateOfRegistration);
    }
    public int getEngineSize()
    {
       return(engineSize);
    }
    public float getPurchasePrice()
    {
       return(purchasePrice);
    }
}
```

Instantiating a class

To use an object of a class type in a program the object must first be instantiated. This means the memory space must be reserved to store the attributes.

The following code instantiates an object `ThisCar` of class `Car`.

Python	`ThisCar = Car("ABC1234", 2500)`
VB.NET	`Dim ThisCar As New Car("ABC1234", 2500)`
Java	`Car thisCar = new Car("ABC1234", 2500);`

Using a method

To call a method in program code, the object identifier is followed by the method identifier and the parameter list.

The following code sets the purchase price for an object `ThisCar` of class `Car`.

Python	`ThisCar.SetPurchasePrice(12000)` `ThisCar.PurchasePrice = 12000 # using properties`
VB.NET	`ThisCar.SetPurchasePrice(12000)` `ThisCar.PurchasePrice = 12000 ' using properties`
Java	`thisCar.setPurchasePrice(12000);`

The following code gets and prints the vehicle ID for an object `ThisCar` of class `Car`.

Python	`print(ThisCar.GetVehicleID())` `print(ThisCar.VehicleID) # using properties`
VB.NET	`Console.WriteLine(ThisCar.GetVehicleID())` `Console.WriteLine(ThisCar.VehicleID) 'using properties`
Java	`System.out.print(thisCar.getVehicleID());`

TASK 27.01

1 Copy the `Car` class definition into your program editor and write a simple program to test that each method works.

2 A business wants to store data about companies they supply. The data to be stored includes: company name, email address, date of last contact.

a Design a class `Company` and draw a class diagram.

b Write program code to declare the class. Company name and email address are to be set by the constructor and will never be changed.

c Instantiate one object of this class and test your class code works.

27.04 Inheritance

The advantage of OOP is that we can design a class (a base class or a superclass) and then derive further classes (subclasses) from this base class. This means that we write the code for the base class only once and the subclasses make use of the attributes and methods of the base class, as well as having their own attributes and methods. This is known as **inheritance** and can be represented by an inheritance diagram (Figure 27.02).

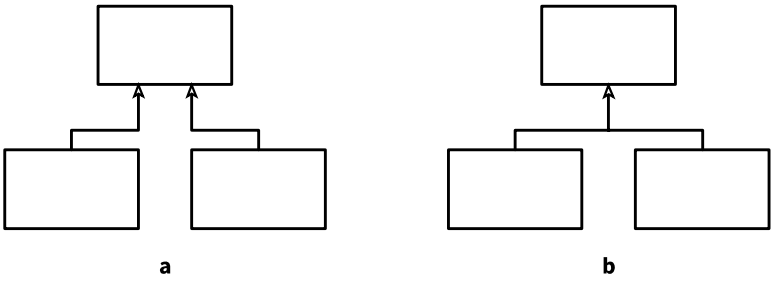


Figure 27.02 Inheritance diagram (a) standard and (b) alternative

WORKED EXAMPLE 27.02

Implementing a library system

Consider the following problem.

- A college library has items for loan.
- The items are currently books and CDs.
- Items can be borrowed for three weeks.
- If a book is on loan, it can be requested by another borrower.

Table 27.02 shows the information to be stored.

Library item	
Book	**CD**
Title of book*	Title of CD*
Author of book*	Artist of CD*
Unique library reference number*	Unique library reference number*
Whether it is on loan*	Whether it is on loan*
The date the book is due for return*	The date the CD is due for return*
Whether the book is requested by another borrower	The type of music on the CD (genre)

Table 27.02 Library system information

The information to be stored about books and CDs needs further analysis. Note that we could have a variable `Title`, which stores the book title or the CD title, depending on which type of library item we are working with. There are further similarities (marked * in Table 27.02).

There are some items of data that are different for books and CDs. Books can be requested by a borrower. For CDs, the genre is to be stored.

We can define a class `LibraryItem` and derive a `Book` class and a `CD` class from it. We can draw the inheritance diagrams for the `LibraryItem`, `Book` and `CD` classes as in Figure 27.03.

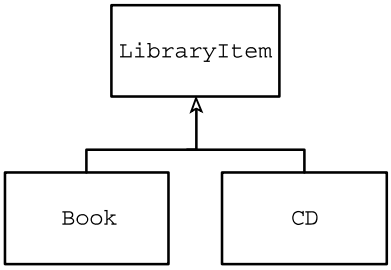


Figure 27.03 Inheritance diagram for LibraryItem, Book and CD classes

Analysing the attributes and methods required for all library items and those only required for books and only for CDs, we arrive at the class diagram in Figure 27.04.

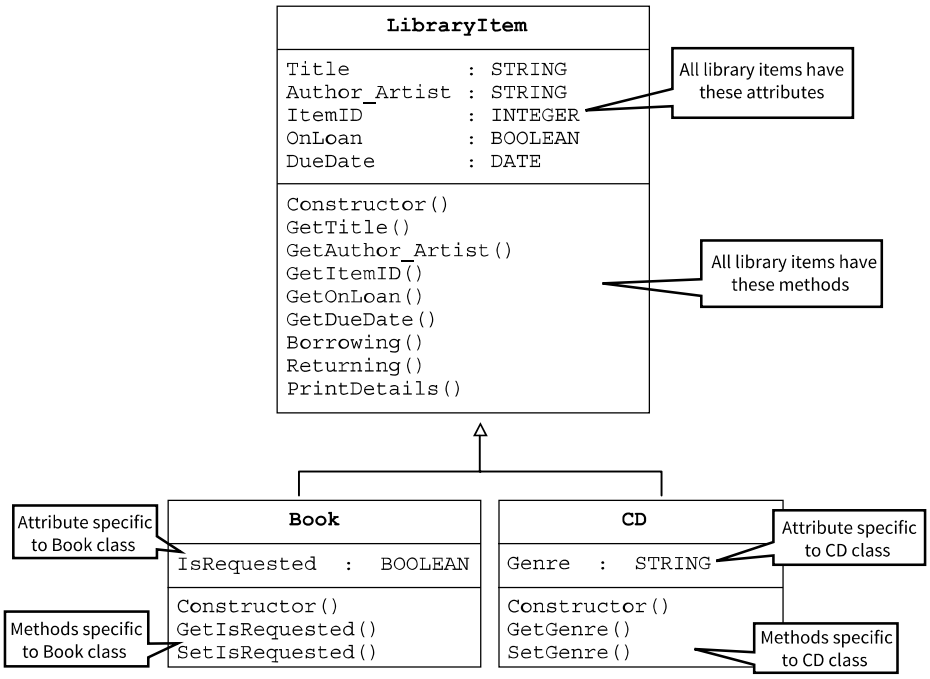


Figure 27.04 Class diagram for LibraryItem, Book and CD

A base class that is never used to create objects directly is known as an **abstract class**. `LibraryItem` is an abstract class.

Declaring a base class and derived classes (subclasses) in Python

The code below shows how a base class and its subclasses are declared in Python.

Python

The base class definition

```
import datetime
class LibraryItem:
    def __ init__(self, t, a, i):            # initialiser method
        self.__Title = t
        self.__Author__Artist = a
        self.__ItemID = i
        self.__OnLoan = False
        self.__DueDate = datetime.date.today()

    def GetTitle(self):
        return(melf.__Title)

# other Get methods go here

    def Borrowing(self):
        self.__OnLoan = True
        self.__DueDate = self.__DueDate + datetime.timedelta(weeks-3)

    def Returning(self):
        self.__OnLoan - False

    def PrintDetails(self):
        print(self.__Title,',', self.__Author__Artist,',')
        print(self.__ItemID',', self.__OnLoan,',',Self,__DueDate)
class Book(Libraryltem):
    def __init__(self, t, a, i):             # initialiser method
        Libraryltem.__init__(self, t, a, i)
        self.__IsRequested = False
        self.__RequestedBy = 0

    def GetIsRequested(self):
        return(self.__IsRequested)

    def SetLiRequested(self):
        self.__IsRequested = True

class CD(LibraryItem):
    def __init__(self, t, a, i):             # initialiser method
        Libraryltem.__init__(self, t, a, i)
        self.__Genre = ''

    def GetGestre(self):
        return(self.__Genre)

    def SetGenre(self, g):
        self.__Genre = g
```

A subclass definition

This statement calls the constructor for the base class

Python using properties

```
class LibraryItem:

    def __init__(self, t, a, i):    # constructor / initialiser method
        self.__Title = t
        self.__Author__Artist = a
        self.__ItemID = i
        self.__OnLoan = False
        self.__DueDate = datetime.date.today()

    @property
    def Title(self):
        return(self.__Title)

# other getters go here

    def Borrowing(self):
        self.__OnLoan = True
        self.__DueDate = self.__DueDate + datetime.timedelta(weeks=3)

    def Returning(self):
        self.__OnLoan = False

    def PrintDetails(self):
        print(self.__Title,'; ', self.__Author__Artist,'; ', end='')
        print(self.__ItemID,'; ', self.__OnLoan,'; ', self.__DueDate)

class Book(LibraryItem):
    def __init__(self, t, a, i):    # initialiser method
        LibraryItem.__init__(self, t, a, i)
        self.__IsRequested = False
        self.__RequestedBy = 0
```

```
    @property
    def IsRequested(self):
        return(self.__IsRequested)

    @IsRequested.setter
    def IsRequested(self, b):
        self.__IsRequested = b

    # print details method for Book

    def PrintDetails(self):
        print("Book Details")
        LibraryItem.PrintDetails(self)
        print(self.__IsRequested)

class CD(LibraryItem):

    def __init __(self, t, a, i):    # initialiser method
        LibraryItem.__init__(self, t, a, i)
        self.__Genre = ""

    @property
     def Genre(self):
        return(self.__Genre)

     @Genre.setter
     def Genre(self, g):
        self.__Genre = g
```

Declaring a base class and derived classes (subclasses) in VB.NET

The code below shows how a base class and its subclasses are declared in VB.NET.

VB.NET

```
Class LibraryItem
    Private Title As String
    Private Author_Artist As String
    Private ItemID As Integer
    Private OnLoan As Boolean = False
    Private DueDate As Date = Today

    Sub Create(ByVal t As String, ByVal a As String, ByVal i As Integer)
        Title = t
        Author_Artist = a
        ItemID = i
    End Sub

    Public Function GetTitle() As String
        Return (Title)
    End Function

    ' other Get method, go here

    Public Sub Borrowing()
        OnLoan = True
        DueDate = DateAdd(DateInterval.Day, 21, Today()) '3 weeks from Today
    End Sub

    Public Sub Returning()
        OnLoan = False
    End Sub

    Sub PrintDetails()
        Console.WriteLine(Title & ", " & ItemID &  ", " & OnLoan &  ", " & DueDate)
    End Sub

End Class

Class Book
    Inherits LibraryItem
    Private IsRequested As Boolean = False
        Public Sub SetIsRequested()
            IsRequested = True
        End Sub

End Class

Class  CD
     Inherits  LibraryItem
     Private  Genre  As String  = ""

     Public  Function  GetGenre() As String
          Return  (Genre)
     End  Function

     Public  Sub  SetGenre(ByVal g As String)
         Genre = g
     End  Sub

End  Class
```

The base class definition

A subclass definition

The Inherits statement is the first statement of a subclass definition

VB.NET using properties

```
class LibraryItem
    Public Property Title()
```

```
    Public Property Author_Artist As String
    Public Property ItemID As Integer
    Public Property OnLoan As Boolean = False
    Public Property DueDate As Date = Today

  Sub Create (ByVal t As String, ByVal a As String, ByVal i As Integer)
     Title = t
     Author_Artist = a
     ItemID = i
   End Sub

  Public Sub Borrowing()
     OnLoan = True
     DueDate = DateAdd(DateInterval.Day, 21, Today()) '3 weeks from today
   End Sub

  Public Sub Returning()
     _OnLoan = False
   End Sub

  Public Sub PrintDetails()
     Console.WriteLine(Title & "; " & ItemID & "; " & OnLoan & "; " &
     DueDate)
   End Sub
End Class

Class Book
   Inherits LibraryItem
   Public Property IsRequested() = False
End Class

Class CD
   Inherits LibraryItem
   Public Property Genre() As String
End Class
```

Declaring a base class and derived classes (subclasses) in Java

The code below shows how a base class and its subclasses are declared in Java.

Java

```
import java.util.Date;
class LibraryItem
{
    private String title;
    private String author_Artist;
    private int itemID;
    private Boolean onLoan;
    private Date dueDate;
    public LibraryItem(String t, String a, int i)  // constructor
    {
       title = t;
       author_Artist = a;
       itemID = i;
       onLoan = false;
       dueDate = new Date();
    }
    public String GetTitle()
    {
       return(title);
    }
// other Get methods go here
    public void borrowing()
    {
       onLoan = true;
       //dueDate = dueDate + 21;
    }
    public void returning()
    {
       onLoan = false;
    }
```

```
    public void printDetails()
    {
       System.out.print(title + " ; " + author_Artist + " ; ");
       System.out.println(itemID + " ; " + onLoan + " ; " + dueDate);
    }
}
class Book extends LibraryItem
{
    private Boolean isRequested;
    private int requestedBy;

    public Book(String t, String a, int i)  // constructor
    {
       super(t, a, i);
       isRequested = false;
       requestedBy = 0;
    }

    public Boolean getIsRequested()
    {
       return(isRequested);
    }
    public void setIsRequested()
    {
       isRequested = true;
    }
}
class CD extends LibraryItem
{
    private String genre;

    public CD(String t, String a, int i)  // constructor
    {
       super(t, a, i);
       genre = "";
    }

    public String getGenre()
    {
       return(genre);
    }

    public void setGenre(String g)
    {
       genre = g;
    }
}
```

Instantiating a subclass

Creating an object of a subclass is done in the same way as with any class (See Section 27.03).

Python

```
ThisBook = Book(Title, Author, ItemID)
ThisCD = CD(Title, Artist, ItemID)
```

VB.NET

```
Dim ThisBook As New Book()
Dim ThisCD As New CD()
ThisBook.Create(Title, Author, ItemID)
ThisCD.Create(Title, Artist, ItemID)
```

Java

```
Book thisBook = new Book("Computing", "Sylvia", 1234);
CD thisCD = new CD("Let it be", "Beatles", 2345);
```

Using a method

Using an object created from a subclass is exactly the same as an object created from any class.

TASK 27.02

Copy the class definitions for `LibraryItem`, `Book` and `CD` into your program editor. Write the additional get methods. Write a simple program to test that each method works.

TASK 27.03

Write code to define a Borrower class as shown in the class diagram in Figure 27.05.

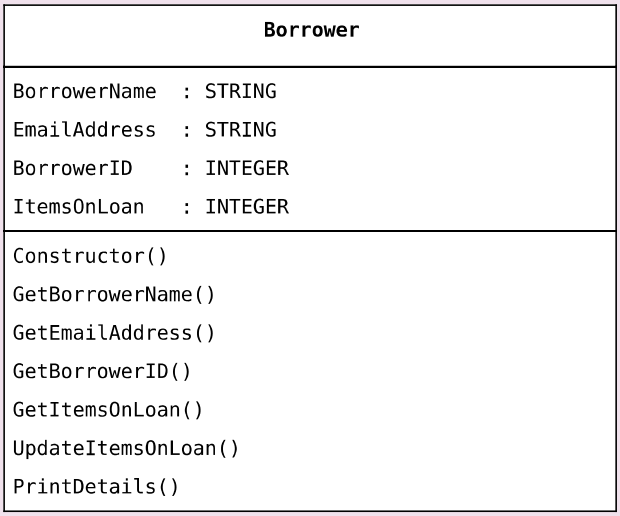


Figure 27.05 Borrower class diagram

The constructor should initialise `ItemsOnLoan` to `0`.

`UpdateItemsOnLoan()` should increment `ItemsOnLoan` by an integer passed as parameter.

Write a simple program to test the methods.

27.05 Polymorphism

Look at Worked Example 27.02 and the code that implements it in Section 27.04. The constructor method of the base class is redefined in the subclasses. The constructor for the `Book` class calls the constructor of the `LibraryItem` class and also initialises the `IsRequested` attribute. The constructor for the `CD` class calls the constructor of the `LibraryItem` class and also initialises the `Genre` attribute.

The `PrintDetails` method is currently only defined in the base class. This means we can only get information on the attributes that are part of the base class. To include the additional attributes from the subclass, we need to declare the method again. Although the method in the subclass will have the same identifier as in the base class, the method will actually behave differently. This is known as **polymorphism**.

The way the programming languages re-define a method varies.

The code shown here includes a call to the base class method with the same name. You can completely re-write the method if required.

Python

```
# define the print details method for Book
def PrintDetails(self):
    print("Book Details")
    LibraryItem.PrintDetails(self)
    print(self.__IsRequested)
```

This line calls the base class method with the same name.

VB.NET

```
' in base class, add the keyword Overridable
' to the method to be redefined
Overridable Sub PrintDetails()

' in subclass, add the redefined method:
Public Overrides Sub PrintDetails()
    Console.WriteLine("Book Details")
    MyBase.PrintDetails()
    Console.WriteLine(IsRequested)

End Sub
```

This line calls the base class method with the same name.

Java

```
@Override
public void printDetails()
{
    System.out.println("Book Details");
    super.printDetails();
    System.out.println(isRequested);
}
```

This line calls the base class method with the same name.

TASK 27.04

Use your program code from Task 27.02. Re-define the `PrintDetail` methods for the `Book` class and the `CD` class. Test your code.

TASK 27.05

Use your program code from Task 27.03. Add another attribute, `BorrowerID`, to the `LibraryItem` class so that the item being loaned can have the borrower recorded.

Change the `LibraryItem.Borrowing` and `LibraryItem.Returning` methods, so that `LoanItem.BorrowerID` and `Borrower.ItemsOnLoan` are updated when a library item is borrowed or returned.

TASK 27.06

Use your code from Task 27.02 or Task 27.04. Add another attribute, `RequestedBy`, to the `Book` class so that the borrower making the request can be recorded.

Change the method `Book.SetIsRequested`, so that `Book.RequestedBy` is updated when a book is requested.

TASK 27.07

Use your code from Task 27.06 to write the complete program to implement a simplified library system.

Write code to provide the user with a menu to choose an option. An example of a menu that would be suitable is shown in Figure 27.06.

```
1 – Add a new borrower
2 – Add a new book
3 – Add a new CD
4 – Borrow a book
5 – Return a book
6 – Borrow a CD
7 – Return a CD
8 – Request book
9 – Print all details
99 – Exit program
Enter your menu choice:
```

Figure 27.06 Library system menu

27.06 Garbage collection

When objects are created they occupy memory. When they are no longer needed, they should be made to release that memory, so it can be re-used. If objects do not let go of memory, we eventually end up with no free memory when we try and run a program. This is known as ‘memory leakage’.

How do our programming languages handle this?

Python	Memory management involves a private heap containing all Python objects and data structures. The management of the Python heap is performed by the interpreter itself. The programmer does not need to do any housekeeping.
VB.NET	A garbage collector automatically reclaims memory from objects that are no longer referred to by the running program.
Java	The Java runtime environment deletes objects when it determines that they are no longer being used.

Table 27.03 Garbage collection strategies

27.07 Containment (aggregation)

In Section 27.04 we covered how a subclass inherits from a base class. This can be seen as generalisation and specialisation. The base class is the most general class, subclasses derived from this base class are more specialised.

We have other kinds of relationships between classes. **Containment** means that one class contains other classes. For example, a car is made up of different parts and each part will be an object based on a class. The wheels are objects of a different class to the engine object. The engine is also made up of different parts. Together, all these parts make up one big object.

The containment relationship is shown in Figure 27.07.

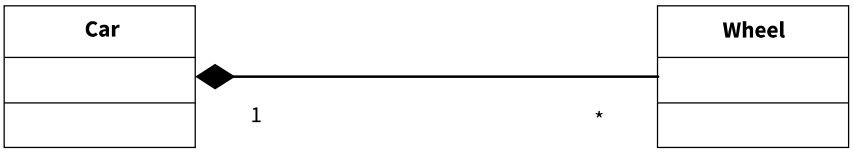


Figure 27.07 Containment (aggregation) class diagram

WORKED EXAMPLE 27.03

Using containment

A college runs courses of up to 50 lessons. A course may end with an assessment. Object-oriented programming is to be used to set up courses. The classes required are shown in Figure 27.08.

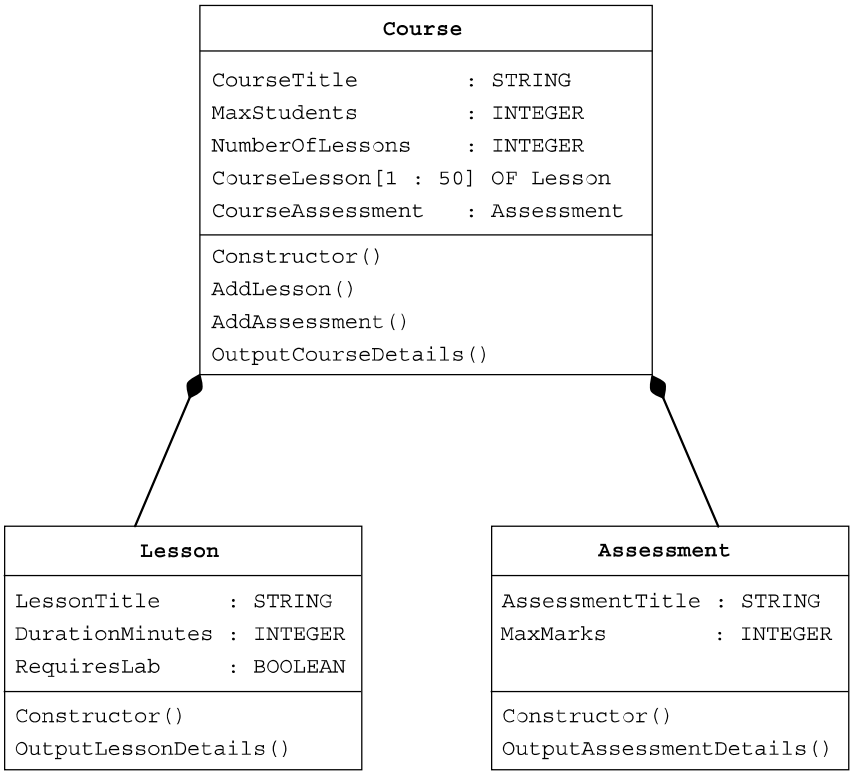


Figure 27.08 Containment class diagram

Assuming that all attributes for the `Lesson` and `Assessment` classes are set by values passed as parameters to the constructor, the code for declaring the `Lesson` and `Assessment` classes is straightforward.

The code below shows how the `Course` class is declared.

Python Course class declaration

```
class Course:
```

```
    def __init__(self, t, m): # sets up a new course
        self.__CourseTitle = t
        self.__MaxStudents = m
        self.__NumberOfLessons = 0
        self.__CourseLesson = []
        self.__CourseAssessment = Assessment

    def AddLesson(self, t, d, r):
        self.__NumberOfLessons = self.__NumberOfLessons + 1
        self.__CourseLesson.append(Lesson(t, d, r))

    def AddAssessment(self, t, m):
        self.__CourseAssessment = Assessment(t, m)

    def OutputCourseDetails(self):
        print(self.__CourseTitle, "Maximum number: ", self.__MaxStudents)
        for i in range(self.__NumberOfLessons):
            print(self.__CourseLesson[i].OutputLessonDetails())
```

VB.NET Course class declaration

```
Class Course
    Private CourseTitle As String
    Private MaxStudents As Integer
    Private NumberOfLessons As Integer = 0
    Private CourseLesson(50) As Lesson
    Private CourseAssessment As Assessment

    Public Sub Create(ByVal t As String, ByVal m As Integer)
        CourseTitle = t
        MaxStudents = m
    End Sub

    Sub AddLesson(ByVal t As String, ByVal d As Integer, ByVal r As Boolean)
        NumberOfLessons = NumberOfLessons + 1
        CourseLesson(NumberOfLessons) = New Lesson
        CourseLesson(NumberOfLessons).Create(t, d, r)
    End Sub
    Public Sub AddAssessment(ByVal t As String, ByVal m As Integer)
        CourseAssessment = New Assessment
        CourseAssessment.Create(t, m)
    End Sub

    Public Sub OutputCourseDetails()
        Console.WriteLine(CourseTitle & "Maximum number: " & MaxStudents)
        For i = 1 To NumberOfLessons
            CourseLesson(i).OutputLessonDetails()
        Next
    End Sub
End Class
```

Java Course class declaration

```
class Course
{
    private String courseTitle;
    private int maxStudents;
    private int numberOfLessons;
```

```
    private Lesson[] courseLesson;
    private Assessment courseAssessment;

    public Course(String t, int m)   // sets up a new course
    {
        courseTitle = t;
        maxStudents = m;
        numberOfLessons = 0;
        courseLesson = new Lesson[50];
        //courseAssessment = new Assessment();
    }
    public void addLesson(String t, int d, Boolean r)
    {
        numberOfLessons = numberOfLessons + 1;
        courseLesson[numberOfLessons] = new Lesson(t, d, r);
    }

    public void addAssessment(String t, int m)
    {
        courseAssessment = new Assessment(t, m);
    }
    public void outputCourseDetails()
    {
        System.out.println(courseTitle + " - Maximum number of students: " + maxStudents);
        for (int i = 1; i < numberOfLessons; i++)
        {
           courseLesson[i].outputLessonDetails();
        }
    }
}
```

Here are simple test programs to check it works.

Python test program

```
def Main():
    MyCourse = Course("Computing", 10) # sets up a new course

    MyCourse.AddAssessment("Programming", 100) # adds an assignment
    # add 3 lessons

    MyCourse.AddLesson("Problem Solving", 60, False)
    MyCourse.AddLesson("Programming", 120, True)
    MyCourse.AddLesson("Theory", 60, False)

    # check it all works
    MyCourse.OutputCourseDetails()
```

VB.NET test program

```
Dim MyCourse As New Course
MyCourse.Create("Computing", 10) ' sets up a new course

MyCourse.AddAssessment("Programming", 100) ' adds an assessment
' add 3 lessons
MyCourse.AddLesson("Problem Solving", 60, False)
MyCourse.AddLesson("Programming", 120, True)
MyCourse.AddLesson("Theory", 60, False)
```

```
'check it all works
MyCourse.OutputCourseDetails()
Console.ReadLine()
```

Java test program

```
Course myCourse = new Course("Computing", 10); // sets up a new course
myCourse.addAssessment("Programming", 100); // adds an assessment
// add 3 lessons
myCourse.addLesson("Problem Solving", 60, false);
myCourse.addLesson("Programming", 120, true);
myCourse.addLesson("Theory", 60, false);
// check it all works
myCourse.outputCourseDetails();
```

TASK 27.08

Write the code required for the `Lesson` and `Assessment` classes. Add the code for the `Course` class and test your program with the appropriate simple program from Worked Example 27.03.

Reflection Point:

Explain why a class can be regarded as a use-defined type.

Summary

- A class has attributes (declared as private) and methods (declared as public) that operate on the attributes. This is known as encapsulation.
- Python and VB.NET support properties: attributes that also include getters and setters.
- A class is a blueprint for creating objects.
- An object is an instance of a class.
- A constructor is a method that instantiates a new object.
- A class and its attributes and methods can be represented by a class diagram.
- Classes (subclasses) can inherit from another class (the base class or superclass). This relationship between a base class and its subclasses can be represented using an inheritance diagram.
- A subclass has all the attributes and methods of its base class. It also has additional attributes and/or methods.
- Polymorphism describes the different behaviour of a subclass method with the same name as the base class method.
- Containment is a relationship between two classes where one class has a component that is of the other class type. This can be represented using a containment diagram.

Exam-style Questions

1 A program is to be written using an object-oriented programming language. A bank account class is designed. Two subclasses have been identified:

- **Personal Account**: the account holder pays a monthly fee and may overdraw the account up to an agreed overdraft limit.
- **Savings Account**: the account holder must maintain a positive balance and gets paid interest on the balance at an agreed interest rate.

a Draw an inheritance diagram for these classes. [3]

The design for the `BankAccount` class consists of:

- attributes:
  - `AccountHolderName`
  - `IBAN`: International Bank Account Number
- methods:
  - `CreateNewAccount`
  - `SetAccountHolderName`
  - `GetAccountHolderName`
  - `GetIBAN`

b Write **program code** for the class definition of the superclass `BankAccount`. [5]

c **i** State the attributes and/or methods required for the subclass `PersonalAccount`. [4]

ii State the attributes and/or methods required for the subclass `SavingsAccount`. [4]

iii Identify the feature of object-oriented program design that combines the attributes and methods into a class. [1]

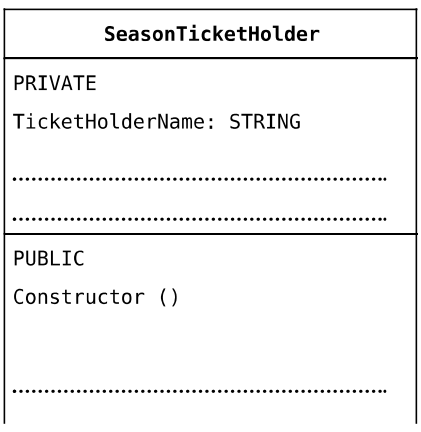
2 A bus company in a town has two types of season ticket for their regular customers: pay-as-you go and contract. All season ticket holders have their name and email address recorded.

A pay-as-you-go ticket holder pays a chosen amount into their account. Each time the ticket holder makes a journey on the bus, the price of the fare is deducted from the amount held in the account. They can top up the amount at any time.

A contract ticket holder pays a fixed fee per month. They can then make unlimited journeys on the bus.

The bus company wants a program to process the season ticket data. The program will be written using an object-oriented programming language.

a Complete the class diagram showing the appropriate attributes and methods. [7]



.....
.....
.....
.....

Pay–As–You–Go–TicketHolder
.....
.....
.....
.....
.....
.....

ContractTicketHolder
.....
.....
Constructor (Name: STRING, email : STRING, Fee : CURRENCY)
.....
.....

b Attributes and methods can be declared as either public or private.

i Explain why the `SeasonTicketHolder` attributes are declared as private. [2]

ii Explain why the `SeasonTicketHolder` methods have been declared as public. [2]

c Write program code to create a new instance of `ContractTicketHolder` with:

- Identifier: `NewCustomer`
- name: A. Smith
- email address: xyz@abc.xx
- monthly fee: $10 [3]

3 A queue abstract data type (ADT) is to be implemented using object-oriented programming. Two classes have been identified: `Queue` and `Node`. The class diagrams are as follows:

QueueClass
Queue : ARRAY[0 : 50] OF NodeClass Head : INTEGER Tail : INTEGER
Constructor() JoinQueue(NewItem : NodeClass) LeaveQueue() : STRING

NodeClass
Data : STRING Pointer : INTEGER
Constructor() SetData(d : STRING) SetPointer(x : INTEGER) GetData() : STRING GetPointer() : INTEGER

a State the relationship between these two classes. [1]

b The `NodeClass` constructor is to:

- create a new node
- initialise the `Data` attribute to the empty string
- initialise the `Pointer` attribute to -1.

Write program code to define `NodeClass`, including the get and set methods. [10]

c The `QueueClass` constructor is to:

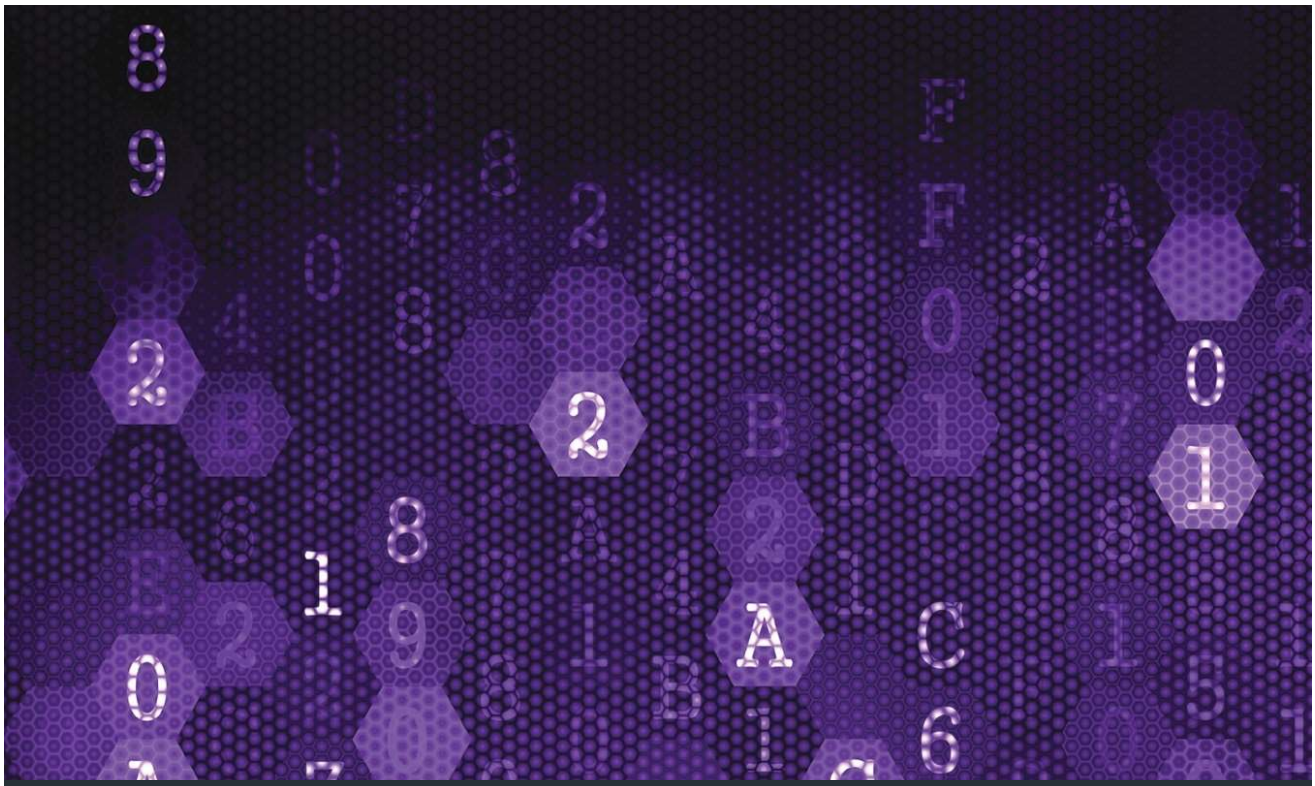
- create a new queue
- initialise the `Head` and `Tail` attributes to -1.

Write program code to define the constructor for `QueueClass`. [3]

d The `JoinQueue` method is to:

- create a new object, `Node`, of `NodeClass`
- assign the value passed as parameter to the `Data` attribute of `Node`
- assign `Node` to the end of `Queue`.

Write program code to define the `JoinQueue` method. [5]

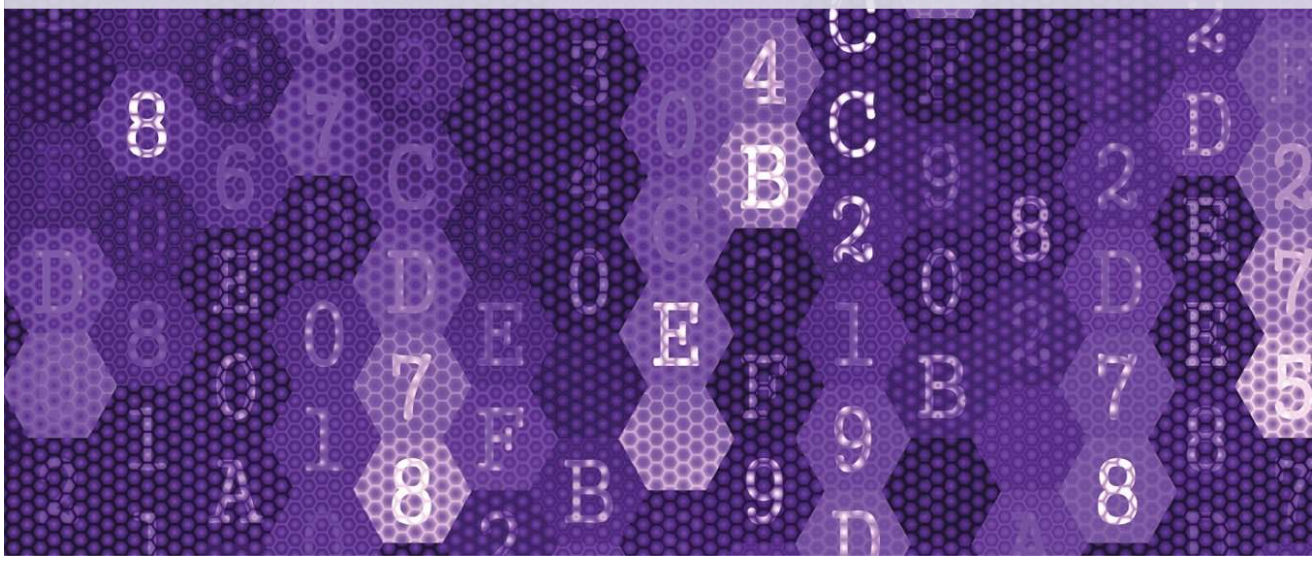


Chapter 28:
Low-level programming

Learning objectives

By the end of this chapter you should be able to:

- write low-level code that uses various address modes: immediate, direct, indirect, indexed and relative.



28.01 Processor instruction set

The instruction set we will use in this chapter is shown in Table 28.01.

Instruction			Explanation
Label	**Op code**	**Operand**	
Data movement instructions			
	LDM	#n	Immediate addressing. Load the denary number *n* to ACC
	LDD	<address>	Direct addressing. Load the contents of the location at the given address to ACC
	LDI	<address>	Indirect addressing. The address to be used is at the given address. Load the contents of this second address to ACC
	LDX	<address>	Indexed addressing. Form the address from <address> + the contents of the index register. Copy the contents of this calculated address to ACC
	LDR	#n	Immediate addressing. Load the denary number *n* to IX
	MOV	<register>	Move the contents of ACC to the given register (IX)
	STO	<address>	Store the contents of ACC at the given address
	STX	<address>	Indexed addressing. Form the address from <address> + the contents of the index register. Copy the contents from ACC to this calculated address*
	STI	<address>	Indirect addressing. The address to be used is at the given address. Store the contents of ACC at this second address*
*Note: The STX and STI instructions are not given as part of the instruction set in the syllabus for 9618			
Arithmetic operations			
	ADD	<address>	Add the contents of the given address to the ACC
	ADD	#n	Add the denary number *n* to the ACC
	SUB	<address>	Subtract the contents of the given address from the ACC
	SUB	#n	Subtract the denary number *n* from the ACC
	INC	<register>	Add 1 to the contents of the register (ACC or IX)
	DEC	<register>	Subtract 1 from the contents of the register (ACC or IX)
Comparison and jump instructions			
	JMP	<address>	Jump to the given address
	CMP	<address>	Compare the contents of ACC with the contents of <address>
	CMP	#n	Compare the contents of ACC with the denary number *n*

	CMI	<address>	Indirect addressing. The address to be used is at the given address. Compare the contents of ACC with the contents of this second address
	JPE	<address>	Following a compare instruction, jump to <address> if the compare was True
	JPN	<address>	Following a compare instruction, jump to <address> if the compare was False
Input/output instructions			
	IN		Key in a character and store its ASCII value in ACC
	OUT		Output to the screen the character whose ASCII value is stored in ACC
Bit manipulation instructions			
	AND	#n	Bitwise AND operation of the contents of ACC with the operand
	AND	<address>	Bitwise AND operation of the contents of ACC with the contents of <address>
	XOR	#n	Bitwise XOR operation of the contents of ACC with the operand
	XOR	<address>	Bitwise XOR operation of the contents of ACC with the contents of <address>
	OR	#n	Bitwise OR operation of the contents of ACC with the operand
	OR	<address>	Bitwise OR operation of the contents of ACC with the contents of <address>
	LSL	#n	Bits in ACC are shifted logically n places to the left. Zeros are introduced on the right-hand end
	LSR	#n	Bits in ACC are shifted logically n places to the right. Zeros are introduced on the left-hand end
Other			
	END		Return control to the operating system
<label>:	<opcode>	<operand>	Labels an instruction
<label>:	<data>		Gives a symbolic address <label> to the memory location with contents <data>

Table 28.01 Processor instruction set

In the assembly code in this chapter:

- ACC denotes the Accumulator
- IX denotes the Index Register
- # denotes immediate addressing
- B denotes a binary number, e.g. B01001010
- & denotes a hexadecimal number, e.g. &4A
- <address> can be an absolute address or a symbolic address.

An instruction set was introduced in Chapter 6.

28.02 Symbolic addresses

A label is a symbolic name for the memory location that it represents. You can treat it like a variable name. When writing low-level programs, we can give absolute addresses of memory locations. This is very restrictive, especially if we want to change the program by adding extra instructions. Writing low-level instructions using symbolic addresses (labels), allows us to think at a higher level. The assembler will allocate absolute addresses during the assembly process (see Chapter 6, Section 6.03).

28.03 Problem-solving and assembly-language programs

When writing a solution to a problem using low-level programming, we need to break down the problem into simple steps that can be programmed using the instruction set available.

One approach is to think in terms of the basic constructs we discussed for high-level languages. You can use the following examples as design patterns.

Assignment

Table 28.02 shows some examples of assembly language assignments that match the pseudocode.

Pseudocode examples	Assembly code examples	Explanation
`A ← 34`	`LDM #34` `STO A`	To store a value in a memory location, the value must first be generated in the accumulator

`B ← B + 1`	`LDD B`
	`INC ACC`
	`STO B`
		To increment the value stored at a memory location: first load the value into the accumulator, increment the value and then store the contents of the accumulator back to the memory location

`B ← B + A`	`LDD B`
	`ADD A`
	`STO B`
		To calculate a value: load the first value from a memory location into the accumulator, then add the value stored at the second memory location to the accumulator and then store the contents of the accumulator to the required memory location

`A ← -A`	`LDD A`
	`XOR #&FF`
	`INC ACC`
	`STO A`
	Alternative method:
	`LDM #0`
	`SUB A`
	`STO A`
		Load the value (assuming eight bits),
		`XOR` with `11111111` to produce the one's complement. Add 1 to get the two's complement.
		Alternatively, set the ACC to zero and then subtract the value from the ACC and store it back in the original address

Table 28.02 Using assignment instructions

TASK 28.01

Write assembly code instructions for this sequence of pseudocode statements:

```
A ← 2

B ← 10

C ← A + B

D ← A - B
```

Selection

Table 28.03 shows some examples of assembly language selections that match the pseudocode.

Pseudocode examples	Assembly code examples	Explanation

`IF A = 0`
`  THEN`
`  B ← B + 1`
	`LDD A`
	`CMP #0`
	`JPN ENDIF`
		Load the contents of the memory location to be tested. Compare it with the required value (in this example 0). If the comparison result

`ENDIF`	`THEN: LDD B`
	`INC ACC`
	`STO B`
	`ENDIF: ...`
		is false (A does not equal 0), a jump over the `THEN` part is required; if the comparison result is true (A = 0) then the following instructions are executed. For ease of understanding, the labels `THEN` and `ENDIF` are used.

`IF A = B`
`THEN`
`OUTPUT "Y"`
`ELSE`
`OUTPUT "N"`
`ENDIF`
	`LDD A`
	`SUB B`
	`CMP #0`
	`JPN ELSE`
	`THEN: LDM #89`
	`OUT`
	`JMP ENDIF`
	`ELSE: LDM #78`
	`OUT`
	`ENDIF: ...`
		Load the contents of A.
		Subtract B.
		If the result is zero, A = B.
		If the comparison result is false (A does not equal B), jump to the ELSE part; if the comparison result is true
		(A = B) the instructions following the `THEN` label are executed.
		Note that a jump over the ELSE part is required.

Table 28.03 Using selection instructions

TASK 28.02

Write assembly code instructions for this sequence of pseudocode statements:

```
IF A <> 0
  THEN
    B ← A
  ELSE
    B ← B - 1
```

Repetition

Table 28.04 shows an example of repetition in assembly language that matches the pseudocode.

Pseudocode example	Assembly code example	Explanation

`A = 0`
`REPEAT`
`OUTPUT "*"`
`A ← A + 1`
`UNTIL A = 5`
	`LDM #0`
	`STO A`
	`LOOP: LDM #42`
	`OUT`
	`LDD A`
	`INC ACC`
	`STO A`
	`CMP #5`
	`JPN LOOP`
		Store the initial value of the counter in A.
		Generate the ASCII code for the character `"*"` and output it.
		Load the counter value,
		increment the counter,
		save it,
		test for final value.
		If final value has not been reached, jump back to beginning of loop.

Table 28.04 Using repetition instructions

TASK 28.03

Write assembly code instructions for this sequence of pseudocode statements:

```
Number ← 1
Total ← 0
```

```
Max ← 5
REPEAT
    Total ← Total + Number
    Number ← Number + 1
UNTIL Number = Max
```

Input/output

Table 28.05 shows some examples of input and output in assembly language that match the pseudocode.

Pseudocode examples	Assembly code examples	Explanation
`INPUT A`	`IN` `STO A`	Store a character input from the keyboard at memory location A.

`OUTPUT B`	`LDM #-1`
	`MOV IX`
	`LOOP: INC IX`
	`LDX B`
	`OUT`
	`CMP #13`
	`JPN LOOP`
		To output a string of characters stored in consecutive locations, starting at location labelled B, a loop and indexed addressing are used.
		The first time round the loop the index register is 0 and the character in memory location B will be loaded into the accumulator and output to the screen. Then a check is made for the end of the string (the carriage return character with ASCII code 13). If it is not the end of the string, jump back to the beginning of the loop.
		To output a number, the number must first be changed into its equivalent string and stored in consecutive memory locations. Then the above method can be used.

`INPUT A`	`LDM #-1`
	`MOV IX`
	`LOOP: INC IX`
	`IN`
	`STX A`
	`CMP #13`
	`JPN LOOP`
		Store a string of characters input from the keyboard into consecutive memory locations starting from A.
		Note: The STX instruction is not given as part of the instruction set in the syllabus for 9618 (from now on referred to as the 9618 instruction set). This means a string can not be saved with the 9618 instruction set.

Table 28.05 Using input and output instructions

TASK 28.04

Write assembly code instructions for this sequence of pseudocode statements:

```
Count ← 0
REPEAT
    Count ← Count + 1
    INPUT Character
UNTIL Character = "N"
```

TASK 28.05

Modify your assembly code instructions from Task 28.04 to implement this sequence of pseudocode statements:

```
Count ← 0
REPEAT
    Count ← Count + 1
    INPUT Character
UNTIL Character = "N"
OUTPUT Count
```

28.04 Absolute and relative addressing

An absolute address is the numeric address of a memory location. A program using absolute addresses cannot be loaded anywhere else in memory. Some assemblers produce relative addresses, so that the program can be loaded anywhere in memory.

Relative addresses are addresses relative to a base address, for example the first instruction of the program. When the program is loaded into memory the base address is stored in a base register `BR`. Instructions that refer to addresses then use the value in the base register, modified by the offset. For example, `STO [BR] + 10` will store the contents of the accumulator at the address calculated from (contents of the base register) + 10.

Table 28.06 shows an example of instructions using symbolic, relative and absolute addressing.

Symbolic addressing	Offset from base (START)	Relative addressing (base address stored in base register)	Absolute addressing
START: LDM #0	0	LDM #0	100 LDM #0
STO A	1	STO [BR] + 10	201 STO 210
LOOP: LDM #42	2	LDM #42	202 LDM #42
OUT	3	OUT	203 OUT
LDD A	4	LDD [BR] + 10	204 LDD 210
INC ACC	5	INC ACC	205 INC ACC
STO A	6	STO [BR] + 10	206 STO 210
CMP #5	7	CMP #5	207 CMP #5
JPN LOOP	8	JPN [BR] + 2	208 JPN 202
END	9	END	209 END
A: 0	10	0	210 0

Table 28.06 Symbolic, relative and absolute addressing

It is very important that, at the end of the program, control is passed back to the operating system. Otherwise the binary pattern held in the next memory location will be interpreted as an instruction. If the content of that memory location does not correspond to a valid instruction, the processor will crash. The instruction `END` signals the end of the program instructions.

Note: the 9618 instruction set does not include relative (base register) addressing.

28.05 Indirect addressing

Indirect addressing is useful if the memory address to be used in an instruction is changed during the execution of the program.

One example is when programming subroutines to which parameters are passed by reference (this is beyond the scope of this book).

Another use of indirect addressing is for a pointer variable.

WORKED EXAMPLE 28.01

Writing a program for a simple queue

At the top level, we can write the problem using structured English:

```
Add a character to the queue:
1  Store the contents of the accumulator in the memory location
pointed to by the tail pointer.
2  Increment the tail pointer.
Remove a character from the queue:
1  Load contents of the memory location at the head of the queue.
2  Increment the head pointer.
```

Table 28.07 shows an example of instructions that implement the above queue-processing algorithms.

Instruction			Explanation
Label	**Op code**	**Operand**	
JOINQ:	STI	TAILPTR	Store contents of ACC in the memory location pointed to by the tail pointer
	LDD	TAILPTR	Increment the tail pointer
	INC	ACC	
	STO	TAILPTR	
	JMP	ENDQ	
LEAVEQ:	LDI	HEADPTR	Load contents of memory location at the head of the queue
	OUT		Output the character
	LDD	HEADPTR	Increment the head pointer
	INC	ACC	
	STO	HEADPTR	
	JMP	ENDQ	
ENDQ:			
HEADPTR:	QSTART		Pointer to start of queue
TAILPTR:	QSTART		Pointer to next free location in queue
QSTART:	""		Start of memory reserved for queue, currently empty

Table 28.07 Queue processing

Note that the value shown in Table 28.07 at the memory locations labelled HEADPTR and TAILPTR is the address of the start of the memory locations reserved for the queue. As values are added to the

queue, the `TAILPTR` value will increase to point to the memory location at the end of the queue data. When a value is taken from the queue, the `HEADPTR` value will increase to point to the memory location at the head of the queue data.

Note: the 9618 instruction set does not include the STI opcode and a character can not be added to the queue using the 9618 instruction set.

TASK 28.06

Write instructions to reverse a word entered at the keyboard. This requires access to an area of memory treated as a stack.

Reflection Point:

Here are the addressing modes you should be able to use when writing low-level code:

Immediate	
Direct	
Indirect	
Indexed	
Relative	

Tick the box next to each addressing mode you can use easily and place a cross in the boxes next to the addressing modes you have difficulty with.

Summary

- A problem to be solved must be broken down into simple steps that can be programmed using the processor's given instruction set.
- A value must be copied into the accumulator before it can be processed.
- Processing includes:
  - arithmetic: adding, incrementing, decrementing
  - comparison: equal or not equal
  - bitwise operations: AND, OR, XOR, shifting
  - output to screen.
- To set a value in the accumulator it can be:
  - input from the keyboard
  - created using immediate addressing
  - loaded from a memory location using direct, relative, indirect or indexed addressing.
- An address can be absolute (a number) or symbolic (a label).

Exam-style Questions

1 The instruction set of a processor with one general-purpose register, the accumulator, includes the following instructions.

Instruction			**Explanation**
Label	**Op code**	**Operand**	
	`LDD`	`<address>`	Direct addressing. Load the contents of the given address to `ACC`
	`STO`	`<address>`	Store the contents of `ACC` at the given address
	`ADD`	`<address>`	Add the contents of the given address to the `ACC`
	`IN`		Key in a character and store its ASCII value in `ACC`
	`AND`	`<address>`	Bitwise `AND` operation of the contents of `ACC` with the contents of `<address>`
	`LSL`	`#n`	Bits in `ACC` are shifted logically *n* places to the left. Zeros are introduced on the right-hand end
	`END`		Return control to the operating system
`<label>:`	`<data>`		Gives a symbolic address `<label>` to the memory location with contents `<data>`

Key to the above table:

- `ACC` denotes the Accumulator.
- `#` denotes immediate addressing.
- `&` denotes a hexadecimal number, e.g. `&4A`.
- `<address>` can be an absolute address or a symbolic address.

a Explain the operation of the `AND` instruction. [1]

b The ASCII code for '0' is the binary value 00110000. The ASCII code for '1' is the binary value 00110001.

Write an `AND` instruction to convert any numeric digit stored in `ACC` in the form of an ASCII code to its eight-bit binary integer equivalent. [1]

c Write the assembly code instructions to convert a two-digit number keyed in at the keyboard to its BCD representation. Store the result in the memory location labelled Result. [7]

Instruction			**Explanation**
Label	**Op code**	**Operand**	
			Input first digit
			Convert from ASCII to its digit value
			Move to upper nibble
			Store in location Result
			Input second digit
			Convert from ASCII to its digit value
			Combine the two values
			Store result
			End of program
`Mask:`			Mask to convert from ASCII to digit equivalent
`Result:`	`&00`		Memory location for result

Result:	&0D0		Memory location for result

2 A given processor has one general-purpose register, the accumulator `ACC`, and one index register, `IX`. Part of the instruction set for this processor is as follows.

Instruction			**Explanation**
Label	**Op code**	**Operand**	
	`LDM`	`#n`	Immediate addressing. Load the number *n* to `ACC`
	`LDD`	`<address>`	Direct addressing. Load the contents of the given address to `ACC`
	`LDX`	`<address>`	Indexed addressing. Form the address from `<address>` + the contents of the Index Register. Copy the contents of this calculated address to `ACC`
	`LDR`	`#n`	Immediate addressing. Load the denary number *n* to `IX`
	`STO`	`<address>`	Store the contents of `ACC` at the given address
	`STX`	`<address>`	Indexed addressing. Form the address from `<address>` + the contents of the index register. Copy the contents from `ACC` to this calculated address
	`ADD`	`<address>`	Add the contents of the given address to the `ACC`
	`INC`	`<register>`	Add 1 to the contents of the register (`ACC` or `IX`)
	`JMP`	`<address>`	Jump to the given address
	`CMP`	`<address>`	Compare the contents of `ACC` with the contents of `<address>`
	`CMP`	`#n`	Compare the contents of `ACC` with the denary number *n*
	`JPE`	`<address>`	Following a compare instruction, jump to `<address>` if the compare was True
	`JPN`	`<address>`	Following a compare instruction, jump to `<address>` if the compare was False
	`IN`		Key in a character and store its ASCII value in `ACC`
	`END`		Return control to the operating system
`<label>:`	`<op code>`	`<operand>`	Labels an instruction
`<label>:`	`<data>`		Gives a symbolic address `<label>` to the memory location with contents `<data>`

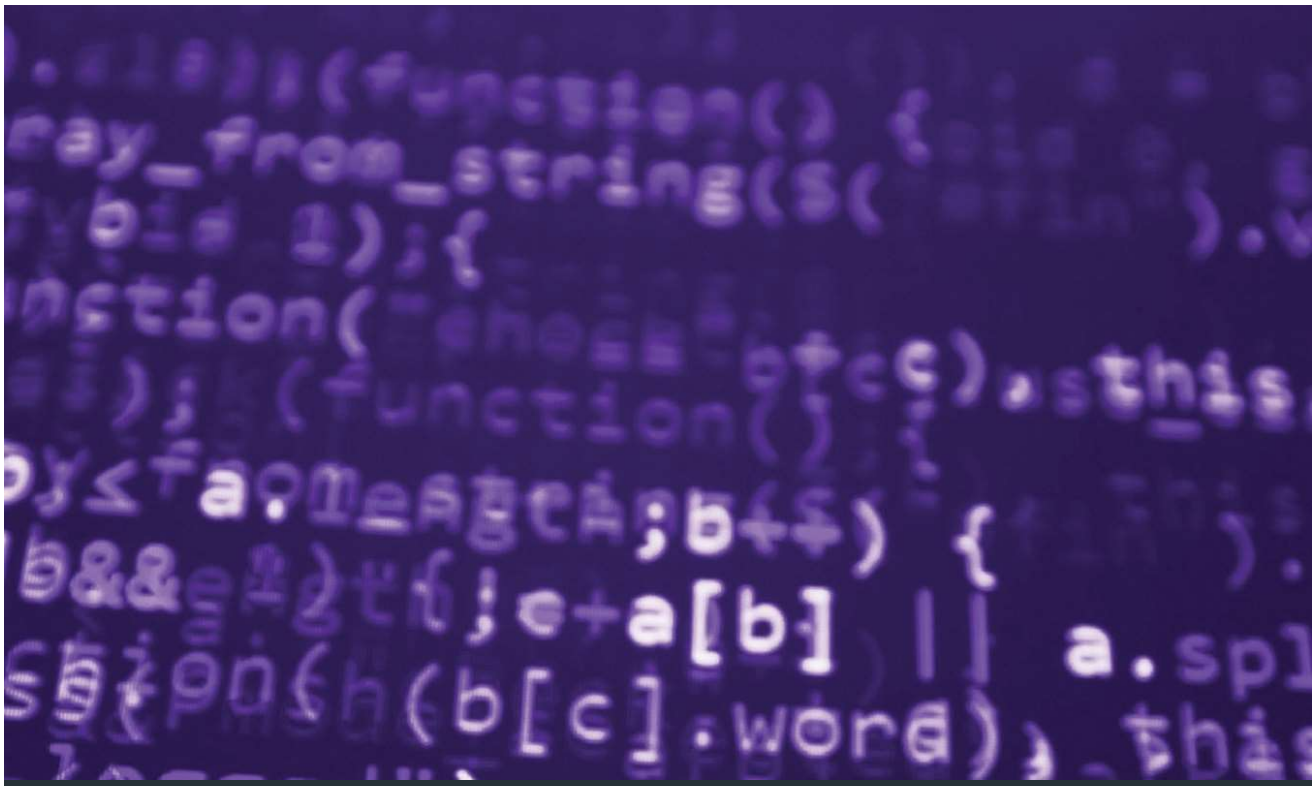
Key to the above table:

- `#` denotes immediate addressing.
- `<address>` can be an absolute address or a symbolic address.

Write an assembly language program that outputs a sequence of characters stored in successive

locations, starting at the location labelled: `STRING`. Output ends when the character in ACC is '!' (ASCII code 33). [8]

Instruction			**Explanation**
Label	**Op code**	**Operand**	
			Set index register to zero
			Load ACC with character stored at STRING (modified by index register)
			Increment index register
			Output character
			Is this character the ! key?
			No – jump to beginning of loop
			End of program
`STRING:`		`72`	String stored from here onwards
		`69`	
		`76`	
		`80`	
		`33`	



Chapter 29:
Declarative programming

Learning objectives

By the end of this chapter you should be able to:

- show understanding of and solve a problem by writing appropriate facts and rules based on supplied information
- show understanding of and write code that can satisfy a goal using facts and rules.

29.01 Declarative programming languages

Declarative languages include database query languages (such as SQL, see Chapter 11, Section 11.07), regular expressions, logic programming and functional programming.

Prolog is a logic programming language widely used for artificial intelligence and expert systems.

The Prolog programs in this chapter have been prepared using the SWI-Prolog environment shown in Figure 29.01 (see SWI-Prolog for a free download).

Figure 29.01 SWI-Prolog environment

29.02 Prolog basics

There are three basic constructs in Prolog: facts, rules and queries.

The program logic is expressed using clauses (facts and rules). Problems are solved by running a query (goal).

A collection of clauses is called a ‘knowledge base’. Writing a Prolog program means writing a knowledge base as a collection of clauses. We use the program by writing queries.

A clause is of the form:

```
Head :- Body.
```

Note that a clause always terminates with a full stop (.)

Prolog has a single data type, called a ‘term’. A term can be:

- an atom, a general-purpose name with no inherent meaning that always starts with a lower case letter
- a number, integer or float (real)
- a variable, denoted by an identifier that starts with a capital letter or an underscore (_)
- a compound term, a predicate, consisting of an atom and a number of arguments in parentheses ().

The arguments themselves can be compound terms. A predicate has an arity (that is, the number of arguments in parentheses).

Prolog is case sensitive.

29.03 Facts in Prolog

A clause without a body is a fact, for example:

```
01  capitalCity(paris).
02  capitalCity(berlin).
03  capitalCity(cairo).
```

The meaning of clause `01` is: Paris is a capital city.
`capitalCity(paris)` is a compound term. Both `capitalCity` and `paris` are atoms.
`capitalCity` is called a predicate and `paris` is the argument.

`capitalCity` has arity 1, as it has just one argument. This can be written as `capitalCity/1`, the `/1` showing that it takes one argument.

TASK 29.01

Launch the editor (File, New ...) from the SWI-Prolog environment. Enter the three clauses, as shown in Figure 29.02. Then save the file (File, Save buffer) as Ex1.

Figure 29.02 Example facts in SWI-Prolog editor

Clauses 01 to 03 are a knowledge base. We can run a query on this knowledge base.

To ask the question whether Paris is a capital city, we write:

```
capitalCity(paris).
```

Prolog answers `true`.

This means: yes, Paris is a capital city.

To ask the question whether London is a capital city, we write:

```
capitalCity(london).
```

Prolog answers `false`.

This means: no, London is not a capital city.

This is because the fact that London is a capital city has not been included in our knowledge base.

TASK 29.02

Run your own queries. You first need to consult the knowledge base (File, Consult ...) from within the Prolog environment. Note that SWI-Prolog uses the prompt ?- (see Figure 29.03).

Figure 29.03 Example queries in SWI-Prolog

If your query does not get a response, check that you have:

- consulted your knowledge base (green text in Figure 29.03)
- used lower-case letters appropriately
- used a full stop at the end of your query.

29.04 Prolog variables

Let's add some more facts to our knowledge base. Comments in Prolog are enclosed in /* and */.

```
04  cityInCountry(paris, france). /* Paris is a city in France */
05  cityInCountry(berlin, germany).
06  cityInCountry(cairo, egypt).
07  cityInCountry(munich, germany).
```

To find out which country Berlin is in, we can run the query (see Figure 29.04):

```
cityInCountry(berlin, Country).
```

Note that `Country` is a variable (it starts with a capital letter).

To find out which cities are in Germany, we can run the query (see Figure 29.04):

```
cityInCountry(City, germany).
```

Figure 29.04 Instantiations of a variable

Note how Prolog responds when running a query that includes a variable. When there is more than one answer, you need to type a semicolon after the first answer and Prolog will give the second answer. The semicolon has the meaning OR. First `City` is instantiated to `berlin` and then `City` is instantiated to `munich`.

WORKED EXAMPLE 29.01

Using a knowledge base

Consider the following knowledge base:

```
01  vegetable(aubergine). /* aubergine is a vegetable */
02  vegetable(potato).
03  vegetable(tomato).
04  meat(chicken). /* chicken is a type of meat */
05  meat(beef).
06  meat(lamb).
07  ingredient(tagine, aubergine, 250). /* tagine contains 250g aubergine */
08  ingredient(tagine, tomato, 100).
09  ingredient(stew, beef, 400).
10  ingredient(stew, potato, 600).
```

We can check the ingredients of a tagine by asking:

```
ingredient(tagine, Ingredient, Amount).
```

Look at the response Prolog gives in Figure 29.05.

Figure 29.05 Instantiation of variables

29.05 The anonymous variable

Consider the knowledge base from Worked Example 29.01. If we are not interested in the amount of each ingredient, we can use the anonymous variable (represented by the underscore character). The query then becomes

```
ingredient(tagine, Ingredient, _).
```

29.06 Rules in Prolog

Remember a clause is of the form `Head :- Body.`

A rule's body consists of calls to predicates, which are called the rule's goals. A predicate is either true or false, based on its terms. If the body of the rule is true, then the head of the rule is true too.

WORKED EXAMPLE 29.02

Using rules in a knowledge base

Consider the following knowledge base:

```
01  parent(fred, jack). /* Fred is the father of Jack */
02  parent(fred, alia).
03  parent(fred, paul).
04  parent(dave, fred).
```

We know that G is a grandparent of S, if G is a parent of P and P is a parent of S.

We could write this as a rule:

`grandparent(G, S) IF parent(G, P) AND parent(P, S).`

However, in Prolog the `IF` is replaced by `:-` and the `AND` is replaced with a comma:

```
grandparent(G, S) :- parent(G, P), parent(P, S).
```

A person has a sibling (brother or sister) if they have the same parent. We can write this as the Prolog rule:

```
sibling(A, B) :-
   parent(P, A),
   parent(P, B).
```

If we run the query

```
sibling(jack, X).
```

we get the answers we expect, but we also get the answer that Jack is his own sibling. To avoid this, we modify the query to ensure that A is not equal to B:

```
sibling(A, B) :-
   parent(P, A),
   parent(P, B),
   not(A=B).
```

Question 29.01

What answer do you expect to get from Prolog to the following query?

`sibling(dave, X).`

TASK 29.03

Write a knowledge base for your own family. You can include more predicates, for example:

Predicate	Meaning
`male(fred).`	`Fred is male`
`female(alia).`	`Alia is female`

Write a rule for father.

Test your program.

WORKED EXAMPLE 29.03

Adding a rule to a knowledge base

Using the knowledge base from Worked Example 29.01, we want to know which dishes contain meat. We are not interested how much meat, so we don't need to know the value of the third argument of the predicate `ingredient/3`. We can write the rule:

```
containsMeat(X) :-
   ingredient(X, Meat, _),
   meat(Meat).
```

The query `containsMeat(X).` returns `X = stew.`

29.07 Instantiation and backtracking

Prolog responds to a query with an answer, such as the one in Worked Example 29.03:

```
X = stew.
```

The = sign is not an assignment as in imperative programs. The = sign shows instantiation.

How does Prolog use the knowledge base to arrive at the answers? One way to see exactly what Prolog is doing is to use the graphical debugger.

WORKED EXAMPLE 29.04

Use the knowledge base from Worked Example 29.03. After consulting the knowledge base, start the debugger (Debug, Graphical debugger) from the Prolog environment. Then type: `trace.` and then the goal as shown in Figure 29.06.

Figure 29.06 Switching on the trace facility

The graphical debugger window opens as shown in Figure 29.07.

Figure 29.07 Graphical debugger

Using the space bar you can step through the program. When Prolog gives an answer in the Prolog Environment window, remember to input a semicolon, so that Prolog will go and check for another possible answer.

If you don't use the graphical debugger but type `trace.` you can see the trace in the SWI-Prolog window, as shown in Figure 29.08.

Figure 29.08 SWI-Prolog trace of goal `containsMeat(X)`

The following terminology is used when discussing a trace:

- `Call` is the initial entry to a predicate
- `creep` indicates that Prolog is moving to the next predicate
- `Exit` is a successful return
- `Redo` indicates that the predicate is backed into for another answer
- `Fail` indicates that Prolog can find no more solutions.

29.08 Recursion

Recursion for imperative languages is covered in Chapter 24. Recursion for declarative languages is where a rule is defined by itself, or more precisely, a rule uses itself as a sub-goal.

Let us expand the Family knowledge base from Worked Example 29.02.

We want a rule that defines whether person A is an ancestor of person B. If A is a parent of B, then A is an ancestor of B. Similarly, if person A is the parent of P, who is the parent of B, then A is an ancestor of B. This is true for the parent of a parent of a parent of B. In general, if A is a parent of X and X is an ancestor of B, then A is an ancestor of B. We can write this information as the rules shown in Figure 29.09.

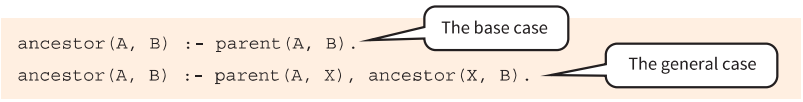


Figure 29.09 Recursive rules

Note that recursion in declarative programming must follow the equivalent rules that imperative programming must follow. A recursive rule must:

- have a base case
- have a general case
- reach the base case after a finite number of calls to itself.

TASK 29.04

Add the ancestor rules to the Family knowledge base and check that the following query gives the correct results:

```
ancestor(A, jack).
```

WORKED EXAMPLE 29.05

Creating the factorial function in Prolog

In Chapter 24, Worked Example 24.01, we programmed the factorial function using recursion with imperative programming. We can also program this function using recursion in Prolog.

```
factorial(0, 1).                    /* base case: 0! = 1    */
factorial(N, Result) :-             /* Result = N!          */
   M is N - 1,                      /* assign N-1 to M      */
   factorial(M, PartResult),        /* PartResult  = (N-1)! */
   Result is PartResult * N.        /* Result = N * (N-1)!  */
```

TASK 29.05

Enter the code from Worked Example 29.05 into the Prolog editor. Save it and consult it. Then pose the following query:

```
factorial(5, Answer).
```

Do you get the correct answer?

29.09 Lists

A list is an ordered collection of terms. It is denoted by square brackets with the terms separated by commas or in the case of the empty list, []. For example [1, 2, 3] or [red, green, blue]. An element can be any type of Prolog object. Different types can be mixed within one list. Lists are used in Prolog where arrays may be used in procedural languages.

Any non-empty list can be thought of as consisting of two parts: the head and the tail. The head is the first item in the list; the tail is the list that remains when we take the first element away. This means that the tail of a list is always a list.

Lists are manipulated by separating the head from the tail. The separator used is a vertical line (a bar):

If Prolog tries to match `[H|T]` to `[car, lorry, boat, ship]`, it will instantiate `H` to `car` and `T` to `[lorry, boat, ship]`.

The clause definition `showHeadAndTail([H|T], H, T).` can be used to pose the query:

```
showHeadAndTail([fred, jack, emma], Head, Tail).
```

Prolog responds with:

```
Head = fred,
Tail = [jack, emma].
```

The clause definition `myList([1,2,3]).` can be used to pose the query:

```
myList([H|T]).
```

Prolog responds with:

```
H = 1,
T = [2, 3].
```

The clause definition `emptyList(A) :- A = [].` can be used to pose the query:

```
emptyList([1]).
```

Prolog responds with:

```
false.
```

List-processing predicate: `append`

The built-in predicate `append(A, B, C)` joins list A and list B and produces list C.

```
append([a, b], [c, d], MyList).
```

produces `MyList = [a, b, c, d].`

```
append(FirstList, [c, d], [a, b, c, d]).
```

produces `FirstList = [a, b].`

List-processing predicate: `member`

The built-in predicate `member(A, B)` returns true if item A is in list B.

```
member(c, [a, b, c, d, e]).
```

produces:

```
true.
```

And

```
member(X,[a, b, c, d]).
```

produces:

```
X = a ;
X = b ;
X = c ;
```

`X = d.`

List-processing predicates: `write` and `read`

The built-in predicate `write(A)` outputs A to the screen.

`write('message: ').` outputs `message: .`

`write(X).` outputs the value currently instantiated with the variable X.

The built-in predicate `read(A)` reads a value from the keyboard into variable A.

`read(Name).` waits for an atom to be input from the keyboard and instantiates the variable `Name` with that value.

Note that the input must start with a lower case letter and not have spaces or be enclosed in quotes.

`nl` moves the output to a new line.

We can write user-friendly programs using the `read` and `write` predicates.

WORKED EXAMPLE 29.06

Using the `read` and `write` predicates

Note how the interface with the user in the code below is written as a rule with the separate steps separated by commas (representing AND).

`assert/1` adds the clause given as the argument to the knowledge base.

`retractall/1` takes the given clause out of the knowledge base, so the next time the program is run, the new facts will be added and used in the goal.

```
/* Weather knowledge base */
weather(good):-
   temp(high),
   humidity(dry),
   sky(sunny).
weather(bad):-
   (humidity(wet);
   temp(low);
   sky(cloudy)).
/* interface */
go:-
   write('Is the temperature high or low? '),
   read(Temp), nl,
   write('Is the sky sunny or cloudy? '),
   read(Sky), nl,
   write('Is the humidity dry or wet? '),
   read(Humidity), nl,
   assert(temp(Temp)),
   assert(sky(Sky)),
   assert(humidity(Humidity)),
   weather(Weather),
   write('The weather is '), write(Weather),
   retractall(temp(_) ),
   retractall(sky(_) ),
   retractall(humidity(_) ).
```

To run the program, type `go.`

TASK 29.06

Test the recursively defined rule writelist/1 to output the elements of a list.

```
writeList([]).
writeList([H|T]):-write(H), nl, writeList(T).
```

Reflection Point:

Can you explain the difference between a fact and a rule?

Summary

- Imperative programs reflect the steps of how to solve a problem.
- Declarative programs reflect what the problem is.
- A knowledge base consists of two types of clause: facts and rules.
- Clauses are sometimes referred to as predicates.
- The arity of a predicate shows how many arguments it takes.
- To solve a problem, the user of the knowledge base poses a query.
- A recursive rule is defined in terms of itself.
- In logic programming, a list is manipulated by separating the head from the tail ([H T]).

Exam-style Questions

1 A logic programming language is used to represent, as a set of facts and rules, details of cities of the world. The set of facts and rules are shown below in clauses labelled 1 to 17.

```
01  capital(vienna).
02  capital(london).
03  capital(santiago).
04  capital(caracas).
05  capital(tokyo).
06  cityIn(vienna, austria).
07  cityIn(santiago, chile).
08  cityIn(salzburg, austria).
09  cityIn(maracaibo, venezuela).
10  continent(austria, europe).
11  continent(chile, southAmerica).
12  continent(uk, europe).
13  continent(argentina, southAmerica).
14  iVisited(vienna).
15  iVisited(tokyo).
16  capitalOf(City, Country)
      IF capital(City) AND cityIn(City, Country).
17  europeanCity(City)
       IF cityIn(City, Country) AND continent(Country, europe).
```

These clauses have the following meanings:

Clause	Meaning
01	Vienna is a capital.
06	Vienna is in Austria.
10	Austria is in the continent of Europe.
14	I visited the city of Vienna.
16	`City` is the capital of `Country` if `City` is a capital and it is in `Country`.
17	`City` is a city in Europe if `City` is in `Country` and `Country` is in Europe.

a Write down the extra clauses needed to express the following facts:

i London is in the UK. [1]

ii I visited the city of Strasbourg. [1]

b The clause `cityIn(City, austria)` would return the result: `vienna, salzburg`.

Write down the result returned by the clause:

```
continent(Country, southAmerica).
```
[2]

c Complete the rule to list countries that I have visited

```
countriesIVisited(Country) IF ...
```
[3]

2 In a particular country, to become a qualified driver you must:

- have a licence: there is a minimum age at which a person can be issued with a licence and it is different for cars and trucks
- pass a theory test: it is the same test for cars and trucks
- pass a driving test for a specific vehicle (car or truck).

A declarative programming language is to be used to represent the knowledge base shown below:

```
01  minimumAge(car, 18).
02  minimumAge(truck, 21).
03  age(fred, 19).
04  age(jack, 22).
05  age(mike, 17).
06  age(jhon, 20).
```

```
07  age(emma, 22).
08  age(sheena, 19).
09  hasLicence(fred).
10  hasLicence(jack).
11  hasLicence(mike).
12  hasLicence(jhon).
13  hasLicence(emma).
14  hasLicence(sheena).
15  allowedToDrive(X, V)
       IF hasLicence(X) AND minimumAge(V, L)
          AND age(X, A)
          AND A >= L.
16  passedTheoryTest(jack).
17  passedTheoryTest(emma).
18  passedTheoryTest(jhon).
19  passedTheoryTest(fred).
20  passedDrivingTest(jhon, car).
21  passedDrivingTest(fred, car).
22  passedDrivingTest(jack, car).
23  passedDrivingTest(jack, truck).
24  passedDrivingTest(sheena, car).
25  qualifiedDriver(X, V)
       IF allowedToDrive(X, V)
          AND passedTheoryTest(X)
          AND passedDrivingTest(X, V).
```

These clauses have the following meanings:

Clause	Meaning
01	The minimum age for a car licence is 18.
03	Fred is aged 19.
09	Fred has a licence.
15	Person `X` is able to drive vehicle `V` if person `X` has a licence and the age `A` of person `X` is greater than or equal to the minimum age `L` to drive vehicle `V`.

a **i** Give **one** example of a fact in this knowledge base. [1]

ii Give **one** example of a rule in this knowledge base. [1]

b State the output produced from these clauses:

i `passedDrivingTest(Who, truck).` [1]

ii `allowedToDrive(mike, car).` [1]

iii `NOT(hasLicence(sheena)).` [1]

c Write a clause to output:

i all qualified car drivers [2]

ii all drivers who have passed the theory test but not a driving test. [3]

d To produce the output from a clause, the inference engine uses a process called backtracking.

Consider the clause:

```
AllowedToDrive(mike, car).
```

Identify the order in which clauses are used to produce the output. For each clause, describe the result that it returns. [5]